Measurement

Without measurements, we never know whether we're moving forward or backward or standing still. It is no coincidence that a major section of the SEI Software Process Assessment questionnaire is devoted to process metrics.

Sample process metrics questions

- Are statistics of software errors gathered?
- Are profiles maintained of actual versus planned test cases for groups completing testing?
- Are software trouble reports resulting from testing tracked to closure?
- Are design and code review coverage measured and recorded?

Mature development and testing organizations are able to provide metrics on all key elements in their process. This is not only because they need to monitor their performance, their use of resources and their efficiency as a basis for management decisions, but also because even asking the questions relating to collecting the statistics is a major springboard for process improvement, at any level.

While measurement can easily look like more unnecessary bureaucracy, there are a number of big questions for testing to which the development of measures within the organization is the only way to get answers, and thus make good decisions in the future.

Measurement can tell us ...

- What was the (real) size of the test effort on a given product?
- How efficient were our verification efforts?
- How thorough were our validation tests?
- What was the quality of the product:
  - during test (before customer use)?
  - in production use by customers?
  - compared to other products?
- How many (approximately) errors are there in the product:
  - before testing begins?
  - later, after some experience with the product?
- When do we stop testing?

Measurement provides an idea of the real program complexity. It helps in planning the test effort, and in predicting how many errors there are and where they occur. Measurement of the number and types of errors detected provides an accurate idea of our efficiency at verification and the weaknesses in our development process.

Measuring validation test coverage provides quantitative assessments of the thoroughness and comprehensiveness of our validation tests. Tracking test execution status monitors the convergence of test categories and provides quantitative information on when to stop testing.

Measuring and tracking incident reports (by severity category):

- provides a leading indicator of product quality;
- provides significant criteria for release readiness;
- correlates to users' satisfaction with the product;
- serves as a predictor of the number of remaining errors;
- when normalized, provides a measure of product quality relative to other products;
- provides a measure of testing efficiency (customer reported vs. testing reported incidents).

Remember that

> "You can't control what you can't measure."
>
> Tom DeMarco, 1982

(See Chapter 12 for more details on useful measures and how to implement them.)

Tools

Testing tools, like any other kinds of tools, provide leverage. There is a wide variety of tools available today to provide assistance in every phase of the testing process. If we are to maximize the benefit we get from tools, there are a number of questions to be asked as part of implementing an effective tools program:

(1) How do the tools fit into and support our test process?
(2) Do we know how to plan and design tests? (Tools do not eliminate the need to think, to plan, and to design.)
(3) Who will be responsible for making sure we get the proper training on our new tool?
(4) Who will promote and support tool use within the organization on an ongoing basis?

In other words, like all testing activities, the use of tools needs to be integrated with the larger picture of our development process.

"Simply throwing a tool at a testing problem will not make it go away."
Dorothy Graham, *The CAST Report*

Tools can be categorized in a number of ways:

- by the testing activity or task in which they are employed (e.g., code verification, test planning, test execution);
- by descriptive functional keyword, in other words the specific function performed by the tool (e.g., capture/reply, logic coverage, comparator);
- by major areas of classification, in other words a small number of high-level classes or groupings of tools, each class containing tools that are similar in function or other characteristic (e.g., test management, static analysis, simulator).

For the purposes of this book, and in order to emphasize that testing has a life cycle of its own, these tools will be discussed in Chapter 11 under the following headings:

- reviews and inspections
- test planning
- test design and development
- test execution and evaluation
- test support.

There is a critical gap between awareness of these tools and effective action in evaluating, selecting, and implementing them. Whatever the level of maturity of the development process, a proper strategy for tool selection and evaluation is essential. (See Chapter 11 for detailed material on tools, their categorization, their acquisition, and implementation.)

What can we do now?

- Raise awareness of standards within your organization. Get copies of one or two key ones. Start with the standard for Software Test Documentation. Demonstrate the benefits of using this standard to assist with writing test plans.
- Make some preliminary evaluations of how you are using tools in your organization. Do you know whether you are doing as well as you might? Access at least one sound objective source of expert advice on tools.

- Obtain a set of SEI Maturity Assessment questions and apply a limited set of them to the relevant part of your organization. Even asking the questions is helpful. It will demonstrate very vividly where you are – and how urgent it is that you start to get some of these disciplines in place. Consider getting independent help from a consultant specializing in process assessment.
- Raise awareness of testware in your organization. It is a capital asset, to be recognized, nurtured, maintained, and expanded. Get management support for testware.
- Find out more about configuration management. Is someone responsible for this in your organization? If it isn't formally implemented, start to use it informally within your own area. Demonstrate the benefits to a wider group.

References

Beresoff, E.H., Henderson, V. D. and Siegel, S.G. (1980). "Software configuration management: a tutorial," *IEEE Tutorial: Software Configuration Management*, IEEE Cat. No. EHO 169-3, 27 October, 24–32.
DeMarco, T. (1982). *Controlling Software Projects*. Yourdon Press.
Humphrey, W.S. (1989). *Managing the Software Process*. Reading, MA: Addison-Wesley.
IEEE/ANSI (1987). IEEE Guide to Software Configuration Management, (Reaff. 1993), IEEE Std 1042-1987.
IEEE/ANSI (1990). IEEE Standard for Software Configuration Management Plans, IEEE Std 828-1990.

The clearing house of Computer Science Technical Reports and source of SEI technical reports is:
Research Access Inc.
3400 Forbes Ave., Suite 302
Pittsburgh, PA 15213
Tel. (1) 800 685 6510

See also:
Software Engineering Institute, Tel. (1) 412 268 7700

PART III
Testing methods

"It is very dangerous to try and leap a chasm in two bounds."

CHINESE PROVERB

Chapter 7
Verification testing

Each verification activity is a phase of the testing life cycle. The testing objective in each verification activity is to detect as many errors as possible. The testing team should leverage its efforts by participating in any inspections and walkthroughs conducted by development and by initiating verification, especially at the early stages of development.

In addition, the testing team should develop its own verification "testware" in the form of generic and testing-specific checklists for all kinds of documents, so that verification expertise becomes an asset of the organization. The testware itself, like all other software, should also be verified.

Testing should use verification practices as a springboard for improving interdisciplinary communication about essential matters as well as promoting the general maturity of the development environment.

You should do as much and as thorough a verification as possible. It has proven to be one of the surest and most cost-effective routes to quality improvement in both the short and the long term.

Basic verification methods

Verification is a "human" examination or review of the work product. There are various types of reviews. Inspections, walkthroughs, technical reviews and other methods are not always referred to consistently, but inspections are generally considered the most formal.

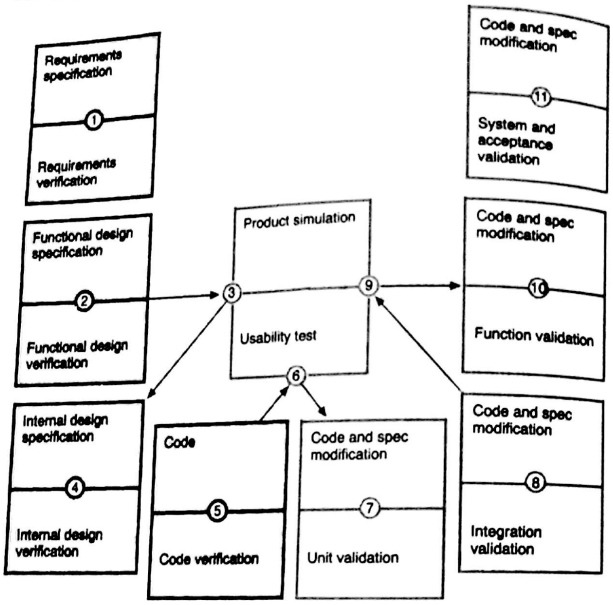


Figure 7.1 The SDT Dotted-U Model, verification testing. (© 1993, 1994 Software Development Technologies)

Basic features of verification methods

	Inspections	Walkthroughs	Buddy checks
Presenter	not author	anyone	none
Participants	team 3–6	larger numbers	1 or 2
Preparation	yes	presenter only	none
Data collected	yes	not required	none
Output report	yes	not required	verbal comment
Advantages	effective	familiarizes larger numbers	inexpensive to do
Disadvantages	short-term cost	finds fewer errors	finds fewer errors

Formal structured types of verification

Formal reviews, technical reviews, and inspections are various expressions used for the more structured types of verification. In the following sections we shall refer mostly to inspections, simply because they are the most structured, but this does not imply the exclusion of the other methods or that we question their value.

The central "event" in these methods is a meeting at which defects in the product under review are discussed and discovered.

Formal methods – key elements

(1) Everyone in the review group participates openly and actively, and participation is governed by traditions, customs and written rules about how such a review is to be conducted.

(2) A written report is produced regarding the status of the product, and the report is available to everyone involved in the project, including management.

(3) The review group is responsible for the quality of information in the written report.

(Freedman and Weinberg, 1990)

Inspections are also characterized by individual preparation by all participants prior to the meeting. Participants are told by the inspection team leader what their reviewer role is, i.e., from what point of view to read the material. When performing inspections well, most of the total defects to be discovered are found during preparation, though an effective inspection meeting will usually uncover some significant additional defects.

Inspection: key elements and phases

Objectives:

- to obtain defects and collect data; meeting does not examine alternative solutions;
- to communicate important work product information.

Elements:

- a planned, structured meeting requiring individual preparation by all participants;
- a team of 3–6 people, led by an impartial moderator;
- presenter is "reader" other than the producer.

Input:
- document to be inspected;
- related source documents;
- general and "tailored" checklists.

Output:
- inspection summary/report;
- data on error types.

Key phases:
- briefing/entry – individual preparation – inspection meeting – edit/rework – exit/re-inspect – collect data;
- some versions of inspection include causal analysis meetings.

The presenter in an inspection meeting is ideally not the producer or author of the document being inspected. Not only does this oblige someone other than the author to look at the product thoroughly, but the meeting audience gets presented with a potentially different interpretation of the material. The other people in the meeting are the inspectors, representing different viewpoints.

Testers participating in any kind of review see the product differently than developers: "How is this thing going to work? If I test it this way, what's going to happen?" As testers we are working out what problems we're going to find when we test it. Our attention is drawn to weak areas of which we might not otherwise have been aware. That is why it is so important that testers are part of the inspection or review process – they bring a unique viewpoint to the meeting.

Any work product that's important can and should be inspected – a project plan, an engineering drawing, a user manual, a test plan. Appendix B includes a generic checklist for use on any document.

Walkthroughs

Walkthroughs are less formal than inspections mainly because of the lack of preparation. In walkthroughs the participants simply come to the meeting; the presenter prepares (usually the author of the product), and there's no additional effort by the participants prior to the meeting.

Walkthroughs: key elements

Objective:
- to detect defects and to become familiar with the material.

Elements:
- a planned meeting where only the presenter must prepare;
- a team of 2–7 people, led by the producer/author;
- the presenter is usually the producer.

Input:
- element under examination, objectives for the walkthrough, applicable standards.

Output:
- report.

Walkthroughs can cover more material than inspections and reviews because the presenter is the producer, and the other participants do not have a heavy participating work load. They therefore provide an opportunity for larger numbers of people to become familiar with the material.

Occasionally, walkthroughs are used for purposes of communication rather than for discovering defects. Software may be "inherited." We don't know exactly what's there so we organize a walkthrough to go through it, page by page, with the key people in the room. If defects are found, that's fine, but the main goal is to familiarize ourselves with the product.

The disadvantage of walkthroughs is that a review tends to be less objective when the presenter is the producer.

From the point of view of finding defects, preparation by participants in inspections and formal reviews usually raise more penetrating issues.

Buddy checks

Any form of human testing, even undisciplined testing, is better than none, provided it is performed by someone other than the author and its objective is to detect defects. There may, for instance, be occasions when it is inappropriate or impossible to get material inspected or formally reviewed.

However, simply giving a document to someone else and asking them to look at it closely will turn up defects we might never find on our own. Some studies show that such desk reviews can be extremely efficient, and they can be a good training ground for finding errors in the product, not the person.

Getting leverage on verification

What (and how much) verification to do?

We have to ask ourselves:

- How many defects are we finding as a result of these reviews?
- How many defects are getting through reviews and getting found in later validation testing?
- What percentage of them are left at the end of testing and only being found by customers?

Verification activities are implemented in an order where the size of the work product, its level of detail, and the cost to verify it are increasing, while the potential payoff is decreasing. This means that, if resources or schedule limitations preclude any of the verification activities (and they almost always will), candidates for elimination should be considered in the reverse order of their occurrence.

Requirements verification offers the biggest potential saving to software development efforts. It can detect many deficiencies that can otherwise go undetected until late in the development cycle, where correction is much more expensive because problems have migrated to other phases. Furthermore, the requirements stage is often where more than 50% of the defects are actually introduced.

In general, formal reviews and inspections are recommended. They are more expensive in the short term, but if done properly, the benefits will always outweigh the costs. They will detect a significant percentage of the errors, and the inherent data collection method is a useful metric for development performance. The success of these methods, and their cost effectiveness, can be measured.

Because they are time consuming and require intense concentration, inspections typically deal with relatively low volumes of material. Like all testing activities, verification of large work products will not be exhaustive and will usually involve risk assessments and trade-offs.

It can be a good idea to "mix and match" verification methods. There are times when it is appropriate to say: "This is the part of the code that's critical to us; it's the heart of the system, so we'll do a proper inspection on it. We're going to get several carefully chosen people and go through every line of the specification." Sampling of important documents can also be useful. Inspecting samples can provide a good estimate of the quality of the document and the number of errors in the uninspected parts of it.

On the other hand we may say: "Here's another part of the code that's not so important. It is not the heart of the system, and we don't want to invest in an inspection." In this case, we may want to do a less formal review or desk check or an informal walkthrough or variations on these. There are always trade-offs, and this is where risk analysis comes into play (see Chapter 5).

Verification is almost always more effective overall than validation testing. It may find defects that are nearly impossible to detect during validation testing. Most importantly, it allows us to find and detect the defects at the earliest possible time.

In most organizations the distribution of verification/validation defects is 20/80, or even less for verification. As a general strategy, we should be working toward a higher proportion of verification, though this can entail a major cultural change. Try verification in one new area, one that shows results

fairly quickly and easily, to get people's support and enthusiasm. Then do a little bit more, slowly shifting the percentages.

Checklists: *the* verification tool

An important tool for verification, especially in more formal forms of verification like inspections, is the checklist. There are generic checklists that can be applied at a high level and maintained for each type of inspection. In other words, there is a checklist for requirements, a checklist for functional design specifications, a checklist for internal design specifications, a checklist for test plans. We can develop our own checklists for anything that we may want to review.

Sample generic checklists

The following checklists are included in Appendix B:

- **Requirements verification checklist**
- **Functional design verification checklist**
- **Internal design verification checklist**
- **Generic code verification checklist (1)**
- **Generic code verification checklist (2)**
- **Code verification checklist for "C"**
- **Code verification checklist for "COBOL"**
- **Generic document verification checklist**

Development and testing often have different checklists. Testing checklists tend to be orientated towards reliability and usability of the product. Development checklists are usually more focused on maintainability and things like guidelines for coding standards.

What is important is to make use of and build on generic checklists, but also to develop our own checklists within our organization for specific purposes and specific projects. These checklists should reflect our chosen focus and our particular present level of maturity in verification testing.

Checklists are an important part of testware. To get maximum leverage on verification, they should be carefully kept, improved, developed, updated – and someone has to take responsibility for this. They are a vital tool for verification testing; they are an important training device; they ensure continuity of the verification effort through different projects and different personnel, and they provide a record of the organization's progress in verification.

Verifying documents at different phases

Verifying requirements

The purpose of the requirements phase is to ensure that the users' needs are properly understood before translating them into design. Requirements are difficult to develop because it is hard to distinguish needs from wants. They will usually change in later phases and have always been the weakest link in the software chain. What do we mean by requirements and what kind of questions are we asking when we do requirements verification?

What's the capability that's needed by the user? What are we trying to provide for the customer? What do they want? The answer is a statement of requirement. The IEEE/ANSI definition is:

> *A requirement is a condition of capability needed by a user to solve a problem or achieve an objective.*
>
> *A requirement is a condition of capability that must be met by or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.*

Requirements may be expressed in formally composed documents or in an informal communication that defines the users' needs. They may be explicit, or implicit, but they are always there, and the purpose of this phase of our testing effort is to ensure that the users' needs are properly understood before we go any further.

One difficulty is to keep requirements strictly separate from solutions to those requirements. These two are frequently confused in documents produced at the early stages of the development process. The other difficulty is to realize that, even in the best of worlds, the requirements are going to change.

In the early days as testers we used to say, "We have to freeze the requirements. We have to get agreement that the requirements will never change, and then make sure that they never do." However, this is unrealistic because in real-world projects requirements do change. What we have to learn is to handle and control the change through proper verification of requirements and configuration management of the various versions (see Chapter 6) and provide ourselves with a solid framework for later testing.

The properties of good requirements specifications

The following are the properties that all good requirements specifications should have:

- visible
- clear (unambiguous)
- complete
- consistent (conflicting requirements must be prioritized)
- reasonable (achievable)
- measurable (quantifiable, testable)
- modifiable
- traceable
- dependent requirements identified.

The IEEE/ANSI guide to software requirements specifications is a very useful reference. It helps customers identify what they want; it helps suppliers understand what they have to provide, and it helps individuals involved with requirements to understand what they're trying to accomplish.

Help from IEEE/ANSI

IEEE/ANSI Standard 830-1993, Recommended Practice for Software Requirements Specifications, helps:

- software customers to accurately describe what they wish to obtain;
- software suppliers to understand exactly what the customer wants;
- individuals to accomplish the following goals:
  - develop standard software requirements specifications outline for their own organizations;
  - define the form and content of their specific software requirements specifications;
  - develop additional local supporting items such as requirements quality checklists or a requirements writers' handbook.

Frequently, the quality of a requirements document is a function of the producer, their skills, whether they know how to write requirements, and whether they are familiar with the standards. The quality of requirements is an important indicator of the level of maturity of an organization. If there is no life cycle, if there is no agreement that there will be a requirements document and there's no agreement on who writes it, what should be in it, or even what are the frameworks for acceptable requirements, development will be difficult and proper testing will be extremely difficult.

When we review requirements, we should be looking for basic functionality, but we need more than that. We may, for instance, need some definitions. Requirements frequently contain important terms and everybody assumes that we know what they mean.

Requirements checklist – sample items

The following is an extract from a generic requirements verification checklist:

- *Precise, unambiguous, and clear*
  Each item is exact and not vague; there is a single interpretation; the meaning of each item is understood; the specification is easy to read.
- *Consistent*
  No item conflicts with another item in the specification.
- *Relevant*
  Each item is pertinent to the problem and its eventual solution.
- *Testable*
  During program development and acceptance testing, it will be possible to determine whether the item has been satisfied.
- *Traceable*
  During program development and testing, it will be possible to trace each item through the various stages of development.

A more complete requirements verification checklist is in Appendix B.

Requirements may make frequent reference to vital, essential issues like security, usability, maintainability, and performance that are entirely non-specific or unexplained. When we're reading requirements from a testability standpoint all this vagueness is revealed for what it is. It's hard to test what constitutes "very good performance." What we can do is come back and say, "Well what did you mean by 'very good'; is that two seconds response time; does it mean 24 hours – or what?" The concept of testability is closely linked to the concept of measurability. Generally, the more quantifiable the requirements, the simpler it is to derive tests for the success of the system that results from them. Testers can provide requirements writers with the framework for the kinds of items that need to be there, like performance specifications, or usability metrics, or maintainability standards.

The testers' viewpoint on requirements is to look for anything that can be a problem. Is it unambiguous, complete, consistent, reasonable? Does it seem achievable? Is it traceable? Is it measurable from a testing standpoint? Verifying requirements provides fertile ground for development improvements as well as later testing. Asking "how can I test this?" will often produce better solutions from product designers as well as ideas about what and how to test.

Exercise on requirements verification

- Demonstrate the effectiveness of requirements verification by trying it out on the requirements document in Appendix C, a simple requirements specification for a reservation system. It can be verified with anything from a one-man "buddy check" to a full-scale inspection. Remember that successful verification takes time.
- Use the requirements verification checklist in Appendix B. Add to the checklist any new items that should be checked in future documents on the basis of this particular verification.
- Record the time taken to do the exercise.
- Estimate the time/cost consequences if the errors found had been allowed to migrate to later stages of development, or even to the user.

Solution: The consolidated notes on this document follows the exercise. It represents four groups of testing practitioners reviewing this document for approximately 30 minutes.

The requirements have been inspected; the changes have been made; the defects have been handled; the requirements have been signed off as approved by the review group. We now have a good basis for test design, for change negotiations and for validation. As a result of participating in requirements verification, we have already learned something about the product and can now plan for validation testing and how it's going to work. We're already ahead of the game!

Verifying the functional design

Functional design is the process of translating user requirements into the set of external (human) interfaces. The output of the process is the functional design specification, which describes the product's behavior as seen by an observer external to the product. It should describe everything the user can see and should avoid describing what the user cannot see. It is eventually translated into an internal design as well as user manuals. It should not include internal information, internal data structures, data diagrams or flow diagrams; they belong in the internal design specification which is the next step in the process.

How is verifying functional design different than verifying the requirements? If requirements are the most important, functional design is the next most important, simply because it is early in the process.

Functional design checklist – sample items

The following is an extract from a generic functional design verification checklist:

- When a term is defined explicitly somewhere, try substituting that definition in place of the term.
- When a structure is described in words, try to sketch a picture of the structure being described.
- When a calculation is specified, work at least two examples by hand and give them as examples in the specification.
- When searching behind certainty statements, *push the search back* as many levels as are needed to achieve the kind of certainty a computer will need.
- Watch for *vague* words, such as *some, sometimes, often, usually, ordinarily, customarily, most,* or *mostly.*

A more complete functional design verification checklist is in Appendix B.

The goal in verifying a functional design is to determine how successfully the user requirements have been incorporated into the functional design. The concept of traceability starts to operate here. We have requirements and we use the requirements document again at this stage as a source document for verifying the functional design. Every paragraph in the requirements should be reflected in the functional design specifications. If it's not, maybe it was dropped completely (but where is this recorded?) or maybe someone simply forgot to implement it.

One of the most common failings of the functional design specifications is incompleteness. Good inspectors or reviewers don't just read what's in front of them. We have to constantly ask: "What's missing?" and keep asking ourselves what should have been written on the page. Try to imagine, if you had been writing the document, what you would have included before you read the section. How should this be described functionally? How should it look to the end user? Pretending to be the designer writing the document enables you to see some of the errors of omission. These are the most important errors to find.

The requirements document itself often has a number of sources, such as standards, correspondence, minutes of meetings, etc. If so, that's where traceability starts. It is also important to look out for unwarranted additions.

Exercise on verifying functional design specification

- Demonstrate the effectiveness of functional design verification by trying it out on the functional design document in Appendix C, a functional design specification for a sales system.
- Use the functional design verification checklist in Appendix B.

Solution: The consolidated notes on this document follows the exercise. It represents four groups of testing practitioners reviewing this document for approximately 30 minutes.

Verifying the internal design

Internal design is the process of translating the functional specification into a detailed set of data structures, data flows, and algorithms. The output of the process is the internal design specification which shows how the product is to be built. Multiple internal design specifications, representing successive levels of abstraction, may be produced. If possible, each of them should be verified.

Internal design checklist – sample items

The following is an extract from a typical internal design verification checklist:

- Does the design document contain a description of the procedure that was used to do preliminary design or is there a reference to such a procedure?
- Is there a model of the user interface to the computing system?
- Is there a high-level functional model of the proposed computing system?
- Are the major implementation alternatives and their evaluations represented in the document?

A more complete internal design verification checklist is in Appendix B.

There is a recommended practice for software design descriptions from IEEE/ANSI in the software engineering standards documentation. It has recommendations for locating information, formats, and ways of organizing the material.

Help from IEEE/ANSI

IEEE/ANSI Standard 1016-1987, IEEE Recommended Practice for Software Design Descriptions (Reaff. 1993), specifies the necessary information content and recommends an organization for software design descriptions. See Appendix A.

Internal design specifications are invaluable to have as a testing perspective. Seeing how the product is going to be built and thinking how the whole system is going to come together enables testers to design additional internals-based tests.

Reviewing internal design involves using checklists, tracing the path back to functional design and back to requirements, and trying to see whether we agree with the algorithms and how they're being put together. We're searching for defects, but we're also thinking, "How would we test that?"

For example, if there's a table that's used in part of the internal design, testers can start asking some questions which are pertinent for testing: "How big is the table? Why are there 25 entries? How do we fill that table? How do we overflow that table? What happens if nothing goes in that table?"

Such questions will provoke good testing ideas. As soon as we start to look at a description of an internal document, the limits within the product become clearer. It shows up boundary conditions; it will warn us about performance and possible failure conditions and all kinds of other internal considerations. From a testing standpoint, whether there are formal internal design specifications or whether we do reviews of them, we can learn a lot about the product by getting this information and thinking about it from a testing perspective.

Verifying the code

Coding is the process of translating the detailed design specification into a specific set of code. The output of the process is the source code itself. This is often the place where companies start when they begin doing walkthroughs and inspections. Sometimes it is the most comfortable place to commence, which is all right as long as we realize it's not the most efficient place to be. Once people are doing walkthroughs and inspections on code, and getting comfortable with the process and learning how to exploit it, we can move them toward reviewing the documents that exist long before there is any code. After all, if we're coding to a poor specification and to the wrong requirements, we've already wasted a lot of time!

Verifying the code involves the following activities:

(1) Comparing the code with internal design specifications.
(2) Examining the code against a language-specific checklist.
(3) Using a static analysis tool to check for compliance with the syntactic/content requirements.
(4) Verifying the correspondence of terms in code with data dictionary and with internal design specification.
(5) Searching for new boundary conditions, possible performance bottlenecks, and other internal considerations which may form the basis for additional validation tests.

Some companies do a formal code review. Participants are given the code in advance; they read through it, look for defects, come to a meeting, walk the code step by step and testers are there helping to find defects and asking for clarification. On the other hand it can be very informal. It can simply be a buddy check, where one person just looks at another person's code and marks the errors in it. On a small scale it can work very effectively.

Code checklist – sample items

The following are typical headings with a single example under each from a generic code verification checklist:

Data reference errors
Is an unset or unitialized variable referenced?

Data declaration errors
Are there variables with similar names?

Computation errors
Is the target variable of an assignment smaller than the right-hand expression?

Comparison errors
Are the conversion rules for comparisons between data or variables of inconsistent type or length handled?

Control flow errors
Is there a possibility of premature loop exit?

Interface errors
If the module has multiple entry points, is a parameter ever referenced that is not associated with the current point of entry?

Input/output errors
Are there grammatical errors in program output text?

Portability
How is the data organized (e.g., packed structures)?

Complete generic and particular code verification checklists are in Appendix B.

We should always be thinking from a testing perspective. Whenever we look at a piece of code, we will think of new tests that we could not have thought of by only reading the requirement or the functional design specification. Often we can spot the error conditions that will provoke failure when the code is first executed. Frequently, developers do straight line checking which does not find these kinds of errors. They will give the code to testing, but the first time the path is executed the whole system crashes.

If code verification is being done more formally, existing checklists in the organization can be used, or a generic checklist can be used as a starting point (see Appendix B) for building a customized version.

Getting the best from verification

The author

People who have their work reviewed in public see themselves in a hot seat, and naturally tend to get defensive. It's important to work hard and consistently for the team spirit attitude: "We are here to find defects in our work products, not to attack individuals for the failures in their part of it." As authors, we should be aware that we are in a position that we may find hard to handle at first, but we have everything to gain from having our work reviewed in this way. Next time around when we're on the team, we should remember to treat the author as we would like to be treated when our work is being reviewed.

As a member of the inspection team, avoid discussions of style. We are there to evaluate the product, not its producer. It's tempting to get diverted into "I would have said it this way," or "It would be much better if you wrote it this way instead of that way." Style issues are not about the content of the information, but the way it has been written. There may, however, be legitimate issues of clarity or definition of terms which ultimately affect the content.

Whatever level of formality is being used in verification, be tactful, be reasonable, and be sensitive to other people's egos. Do any preparation required for inspections properly, and at the inspection meeting, raise issues – don't try to solve them.

If we can develop these attitudes successfully within the testing organization (and acting as an example goes a very long way), we will be the instigators of a critical culture change.

The development team

Reviews and inspections can help with communication and with motivation, sometimes through a sense of pride or even embarrassment. If people know their work product is going to be inspected they tend to do a better job.

Feedback can be very positive. There are times in review meetings when producers/authors get a sense of "that looks good, that's a good way to do it" from the team, and it becomes, without being formally adopted, common good practice within the organization. Feedback sharpens people's level of performance in a constructive way, and in six months to a year, we'll get a reduction in defects, just because we know our material is going to be under scrutiny the next time there is a review.

The other issue that is really important is communication. We don't measure our success in inspection purely by defects that are being found. Inspection is also about improving the software development process. One of the things that contributes to the value of the inspection is that people start communicating about things that they need to know to do their jobs properly.

For example, being on an inspection team enables testers to find out basic information about the product that they didn't have before. This communication value of inspection should never be ignored.

Verification, especially at the early stages, can help with communication not only within the development environment, but throughout the organization. If we are doing requirements verification, for instance, ideally we need the marketing people, as well as the test people and development people there, and maybe support people as well. It's a golden opportunity to spread understanding and better communication to a wider group.

The inspection team

Inspection is difficult because there is no detailed cookbook method for any given type of work product. Each type of work product requires different expertise and thought processes. Critical thinking, which must often transcend even the most robust and well-defined inspection checklists, is required.

Inspection also requires intense concentration and is very fatiguing if overused. It requires the ability to detect omissions. Most people react only to what they see in front of them. The best reviewers and inspectors must ask "What's missing?" or "What should be written here that isn't?" The tough thing is to ask what should have been on the page but isn't there. Good reviewers, especially good inspection team people, are worth their weight in gold in organizations. After a while, everybody knows who they are, and they are constantly in demand for any kind of verification activity.

Cost-effective verification

Verification is just as productive and important as validation testing, if not more so. It has the potential to detect defects at the earliest possible stage, thereby minimizing the cost of correction. But is the verification process itself cost effective? Do we save more than we spend by finding errors early? The short answer is that it has been demonstrated again and again that verification, although it isn't free, is cost effective.

We should therefore try to verify critical documents, or at least some parts of them. On a large work product, we will almost certainly feel we can't do all the ideal complete code reviews and formal specification and design reviews. We may want to use walkthroughs for some parts, instead of having formal reviews, and we may want to do desk checks for some items. These are all decisions based on the particular case.

We can increase the cost effectiveness of verification by having a good configuration management system (see Chapter 6). Somebody must make sure we're not verifying the wrong thing at the wrong time, and that what we verify is the same thing that we validation-test and that we eventually ship.

Three critical success factors for implementing verification

Success factor 1: process ownership

If we are not already using verification or we need to improve the way we are doing it, the process needs a champion. The champion can be anybody who really is interested in it, who cares a lot about it, and who will take ownership for the process and will make it happen. It could be someone in the development organization, who knows inspections are a good thing. It could be a quality assurance person; it could be a process expert, or someone in the process engineering group. It could be a full-time job or it could be a part-time job. When companies get really big they hire people to do nothing but organize formal verification processes. In smaller companies this is neither necessary nor possible. The important thing is that somebody must become the champion of the method and work in the medium and long term to gradually gain support for it.

Success factor 2: management support

Frequently, it is management that initiates the adoption of inspections. Whether or not they are initiating the process, it is important that managers are well briefed on inspections and their benefits. It is unreasonable to expect them to spend resources and support the effort if they don't see the long-term gains. It can be difficult for anyone with different pressures and different immediate problems to solve to understand why it's worth putting resources into these early testing efforts.

You should promote support for inspections by getting out early results. If possible, collect data on the errors found by verifying at the early stages and compare this to the estimated costs if they had been allowed to migrate to the later stages. Get people who aren't so enthusiastic about these methods to a meeting to discuss problems with a particular work product, and then demonstrate with a small, live inspection how effective inspections are in their own organization and on their own material. Show them it's possible to monitor the cost effectiveness of the new process.

Success factor 3: training

Training in reviews and inspections is crucial, including specific training for practitioners on how to perform reviews and inspections, including costs, benefits, and dealing with the human and cultural issues. Training should also include performing inspections in a workshop setting of real work products from the local environment. Everyone who is going to be inspecting documents should be trained. Where this is not possible, an experienced team with a good grasp of the human issues involved can absorb a new member.

Help from SIRO

Software Inspection and Review Organization (SIRO)
This organization was formed to exchange new ideas and information about group-based software examinations. It facilitates emerging inspection and review techniques, provides a clearing house for resources, and surveys/reports on the current use of inspections and reviews.
Contact:
SIRO
PO Box 61015
Sunnyvale, CA 94088-1015

Recommendations

Inspections are recommended first and foremost because they have proven to be the most effective, most practical method of verification. Inspections are probably the oldest methodology in the history of software development. They have been used successfully for at least 25 years. So while inspection techniques have evolved over this time, you won't be suggesting untried leading edge methods when you suggest implementing inspections in your organization. They work. Getting the right people in a room to look at the right material in a systematic, objective way is simply common sense and good basic communication.

There's a real trade-off between verification and validation testing. In doing verification we begin to ease the pain in validation testing. Most testing organizations experience "The Chaos Zone" when they only do functional testing or system testing. At that point, all of the code, of which testing has little or no prior knowledge, is thrown into the testing department. All kinds of defects are found through hasty testing in unfamiliar territory. Shipment dates are missed and pressure is created because testers are operating in an unfamiliar environment where there should be an opportunity to get acquainted with the product and find many defects a lot earlier.

Get started by inspecting some key material. Typically, an organization will have all kinds of projects all at different stages. Some projects may be so far along that it doesn't make sense to implement verification on them.

It is usually better not to go for a "big bang" approach and implement formal inspections on all documentation from a certain point in time. It's best to pick some high-risk material with a high pay-off on a number of different projects first. Perhaps start by reviewing all new requirement specifications or

all new and changed code on critical projects. Build support while measuring and tracking enthusiasm from peers. Then do inspections on a wider range of documentation and demonstrate the results.

Show management or colleagues who don't know about inspections the results of the requirements and functional design verification exercises in this chapter. Be sure to include the time resources used and the estimated consequences of allowing the errors found to migrate to later stages.

References

Bender, D. (1993). "Writing testable requirements," *Software Testing Analysis & Review (STAR) Conference Proceedings*. (This provides a set of practical guidelines for writing testable requirements.)

Fagan, M.E. (1976). "Design and code inspection to reduce errors in program development," *IBM Systems Journal*, 15(3).

Freedman, D.P. and Weinberg, G.M. (1990). *Handbook of Walkthroughs, Inspections, and Technical Reviews*. New York: Dorset.

Gilb, T. and Graham, D. (1993). *Software Inspection*. Wokingham: Addison-Wesley.

IEEE/ANSI (1988). IEEE Standard for Software Reviews and Audits, IEEE Std 1028-1988.

Chapter 8
Validation testing

Validation overview

Let us begin with eight axioms that apply to all validation testing:

(1) Testing can be used to show the presence of errors, but never their absence.
(2) One of the most difficult problems in testing is knowing when to stop.
(3) Avoid unplanned, non-reusable, throw-away test cases unless the program is truly a throw-away program.
(4) A necessary part of a test case is a definition of the expected output or result. Always carefully compare the actual versus the expected results of each test.
(5) Test cases must be written for invalid and unexpected, as well as valid and expected, input conditions. "Invalid" is defined as a condition that is outside the set of valid conditions and should be diagnosed as such by the program being tested.
(6) Test cases must be written to generate desired output conditions. Less experienced testers tend to think only from the input perspective. Experienced testers determine the inputs required to generate a predesigned set of outputs.
(7) With the exception of unit and integration testing, a program should not be tested by the person or organization that developed it. Practical cost considerations usually require developers do unit and integration testing.
(8) The number of undiscovered errors is directly proportional to the number of errors already discovered.

The IEEE/ANSI definition is as follows:

> *Validation* is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

The eight axioms are quite useful, but how do we determine in practice whether a program does in fact meet its requirements? There are two keys:

(1) Developing tests that will determine whether the product satisfies the users' requirements, as stated in the requirements specification.
(2) Developing tests that will determine whether the product's actual behavior matches the desired behavior, as described in the functional design specification.

Even though the internal design and code are derived from the functional design, it is usually not necessary to understand either of them to determine whether the end product meets its requirements.

In IEEE/ANSI usage, note the word "requirements" includes both user requirements and functional interfaces. In practice, these should be created and maintained as two distinct documents. The high-level requirements are written from a customer or market perspective, while the functional specification is written from an engineering perspective. A significant issue in many companies is trying to define a clear boundary between marketing and engineering specification responsibilities. By defining the format and content of these two key documents separately, many organizations manage to obtain agreement on organizational ownership; e.g., marketing is responsible for creating the requirements specification, and software engineering is responsible for the functional design specification (see Figure 8.1).

Coverage

How do we measure how thoroughly tested a product is? What is the measure of "testedness"? To what degree do our test cases adequately cover the product? How do we quantitatively measure how good a job we are doing as testers?

The execution of a given test case against program P will:

- address (cover) certain requirements of P;
- utilize (cover) certain parts of P's functionality;
- exercise (cover) certain parts of P's internal logic.

We have to be sure we have enough tests at each of these levels.

The measures of testedness for P are the degrees to which the collective set of test cases for P enhance the requirements coverage, the function coverage and the logic coverage.

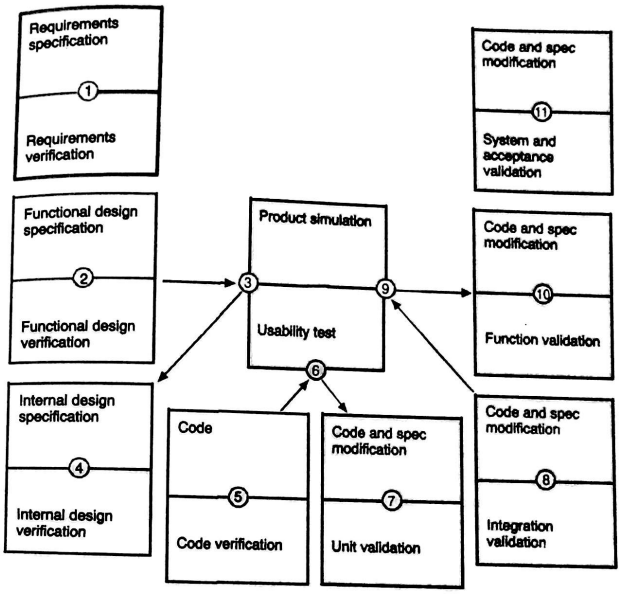


Figure 8.1 The SDT Dotted-U Model, validation. (© 1993, 1994 Software Development Technologies)

Fundamental testing strategies

Black-box testing and white-box testing are the two fundamental testing strategies. They are strategies, not technical or analytical methods.

Black-box tests are derived from the functional design specification, without regard to the internal program structure. Black-box testing tests the product against the end user, external specifications. Black-box testing is done without any internal knowledge of the product. It is important in practice to try to test, or at least write, detailed test plans for requirements and functional specifications tests without too much knowledge of the code. Understanding the code changes the way the requirements are seen, and test design should not be "contaminated" by this knowledge too early.

Black-box testing will not test hidden functions (i.e., functions implemented but not described in the functional design specification), and the errors associated with them will not be found in black-box testing.

White-box tests require knowledge of the internal program structure and are derived from the internal design specification or the code. They will not detect missing functions (i.e., those described in the functional design specification but not supported by the internal specification or code).

Validation mission vs. test coverage

Exhaustive testing is impossible and the testing of any program will be necessarily incomplete. The negative effects of this incompleteness are minimized by identifying the subset of all possible test cases that has the highest probability of detecting the most errors. In this way we can find the greatest possible number of errors with a finite number of tests.

Test coverage has three components: requirements coverage, function coverage, and logic coverage. IEEE/ANSI clearly states the importance of the first two of these, but in fact logic coverage is also very important for two reasons:

(1) it indirectly improves function coverage;
(2) it is necessary for the testing of logic paths that are not discernible from the external functionality (e.g., a math function that uses completely different algorithms, depending on the values of the input arguments).

Coverage is typically referred to at the statement level. The simplest form is the percentage of the statements in the program that are being executed by the test: "80% coverage" means that 80% of the statements in the program have been tested. On a project involving compiler testing, the author discovered that, if by going to 75% coverage, we found x bugs, then by increasing coverage from 75% to 85%, another x bugs will be found. Likewise from 85 to 90% we find two to three times x bugs again! Generally stated, the probability of finding new defects is inversely proportional to the amount of code not yet covered. The closer we are to achieving 100% coverage, the more likely it becomes that we are searching in previously un-navigated territory and the more likely it is that we will find more defects per line of code.

Test basis

The basis of a test is the source material (of the product under test) that provides the stimulus for the test. In other words, it is the area targeted as the potential source of an error:

- Requirements-based tests are based on the requirements document.
- Function-based tests are based on the functional design specification.
- Internals-based tests are based on the internal design specification or code.

Function-based and internals-based tests will fail to detect situations where requirements are not met. Internals-based tests will fail to detect errors in functionality.

Validation strategies

Given the alternatives of black-box and white-box testing, and the need for requirements, function, and logic test coverage, what should our overall strategy be?

- *Requirements-based tests* should employ the black-box strategy. User requirements can be tested without knowledge of the internal design specification or the code. Tests are based on the requirements document but formulated by using the functional design specification.
- *Function-based tests* should employ the black-box strategy. Using the functional design specification to design function-based tests is both necessary and sufficient. The requirements and internal design specifications are unnecessary for function-based tests.
- *Internals-based tests* must necessarily employ the white-box strategy. Tests can be formulated by using the functional design specification.

There are two basic requirements which apply to all validation tests:

(1) definition of results
(2) repeatability.

A necessary part of a test case is a definition of the expected output or result. If the expected result is not pre-defined, it is all too easy to interpret the actual result as correct. Once again, it was Myers who saw the significance of our perceptual and cognitive mechanisms for testing: "The eye sees what it wants to see." (Myers, 1979: p. 12)

The test case should be repeatable. In other words, it should produce identical results each time it is run against the same software/hardware configuration. When actual and expected results disagree, the failure should be able to be recreated by development in its debugging efforts. However, repeatability is not always possible, for example, when the software and/or

hardware is handling asynchronous processes. Here are some IEEE/ANSI definitions:

A *test*:
(i) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.
(ii) A set of one or more test cases.

IEEE/ANSI's second definition of "test" requires additional terms to identify separate and distinct, lower-level tests within a single test case.

A *test case*:
(i) A set of test inputs, execution conditions, and expected results developed for a particular objective.
(ii) The smallest entity that is always executed as a unit, from beginning to end.

A test case may perform any number of discrete *subtests*.

A *test procedure*:
(i) The detailed instructions for the set-up, execution, and evaluation of results for a given test case.
(ii) A test case may be used in more than one test procedure. (See Figure 8.2.)

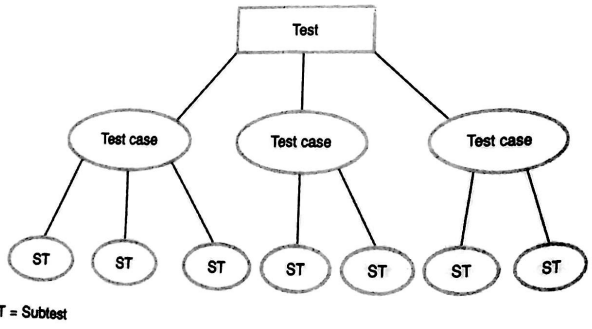


Figure 8.2 Tests, test cases, and subtests. (© 1993, 1994 Software Development Technologies)

Validation methods

Proven test-design methods provide a more intelligent and effective means of identifying tests than a purely random approach.

This section describes some detailed technical and analytical methods for designing high-yield tests. Each method is described in testing literature. The ones outlined here are the most commonly used, practical, and useful. Each has strengths and weaknesses (i.e., types of errors it is likely to detect and those it will fail to detect).

Black-box methods for function-based tests

The following methods are commonly used:

- equivalence partitioning
- boundary-value analysis
- error guessing.

The following are lesser-used methods:

- cause-effect graphing
- syntax testing
- state transition testing
- graph matrix.

Equivalence partitioning

Equivalence partitioning is a systematic process that identifies, on the basis of whatever information is available, a set of interesting classes of input conditions to be tested, where each class is representative of (or covers) a large set of other possible tests. If partitioning is applied to the product under test, the product is going to behave in much the same way for all members of the class.

The aim is to minimize the number of test cases required to cover these input conditions.

There are two distinct steps. The first is to identify the equivalence classes (ECs) and the second is to identify the test cases.

(1) *Identifying equivalence classes*
For each external input:
(i) If the input specifies *a range* of valid values, define one valid EC (within the range) and two invalid ECs (one outside each end of the range).

Example: If the input requires a month in the range of 1–12, define one valid EC for months 1 though 12 and two invalid ECs (month<1 and month>12).

(ii) If the input specifies the *number* (N) of valid values, define one valid EC and two invalid ECs (none, and more than N).
Example: If the input requires the titles of at least three but no more than eight books, then define one valid EC and two invalid ECs (<3 and >8 books).

(iii) If the input specifies a *set* of valid values, define one valid EC (within the set) and one invalid EC (outside the set).
Example: If the input requires one of the names TOM, DICK, or HARRY, then define one valid EC (using one of the valid names) and one invalid EC (using the name JOE).

(iv) If there is reason to believe that the program handles each valid input differently, then define one valid EC per valid input.

(v) If the input specifies a "must be" situation, define one valid EC and one invalid EC.
Example: If the first character of the input must be numeric, then define one valid EC where the first character is a number and one invalid EC where the first character is not a number.

(vi) If there is reason to believe that elements in an EC are not handled in an identical manner by the program, subdivide the EC into smaller ECs.

(2) *Identifying test cases*
(i) Assign a unique number to each EC.
(ii) Until all valid ECs have been covered by test cases, write a new test case covering as many of the uncovered ECs as possible.
(iii) Until all invalid ECs have been covered by test cases, write a test case that covers one, and only one, of the uncovered invalid ECs.
(iv) If multiple invalid ECs are tested in the same test case, some of those tests may never be executed because the first test may mask other tests or terminate execution of the test case.

Reminder: A necessary part of any test case is a description of the expected results, even for tests that use invalid inputs.

Equivalence partitioning significantly reduces the number of input conditions to be tested by identifying classes of conditions that are equivalent to many other conditions. It does not test combinations of input conditions.

Exercise on equivalence partitioning
Equivalence partitioning looks easy, but takes practice. For practice we have included an exercise along with one possible solution. Read the program description below, and then complete the worksheet that follows.

FUNCTIONAL DESIGN SPECIFICATION FOR GOLFSCORE

GOLFSCORE is a program which calculates the scores of the participants in a golf tournament which is based on the following assumptions and scoring rules:

Assumptions:
(1) The number of courses played can be from 1 to 8.
(2) The number of participating golfers can be from 2 to 400.
(3) Each golfer plays each course once.
(4) A golfer's tournament score is the sum of his/her scores on each course.
(5) Each golf course has 18 holes, and par for each hole is 3, 4, or 5.

Scoring rules for each hole:

Strokes	**Score**
over par	0
par	1
1 under par	2
2 under par	4
>2 under par	6

Input
Input to GOLFSCORE is a formatted text file containing the following records, in sequence:

(1) **Course records.** One record for each golf course. Each record contains the name of the course and the par for each of its 18 holes.

Column 1:	Blank
Columns 2–19:	Course name
Columns 21–38:	Par for holes 1–18 (par is an integer 3, 4, or 5)

(2) **Delimiter record.** Denotes the end of the course records.

Column 1:	Non-blank
Columns 2–60:	Blank

(3) **Golfer records.** One record per golfer per course (in any order). Each record contains the name of the golfer, the name of the course, and the actual number of strokes taken for each of the 18 holes.

Column 1:	Blank
Columns 2–19:	Course name
Columns 22–39:	Golfer name
Columns 41–58:	Number of strokes taken for holes 1–18 (per hole, number of strokes is a single, non-zero digit)

(4) **Delimiter record.** Denotes the end of the golfer records.

Column 1:	Non-blank
Columns 2–60:	Blank

Output
GOLFSCORE produces the following output reports, showing:
(1) The names of the golfers, their scores for each course, their total scores, and their final rank in the tournament, sorted in descending order of total score.
(2) The same as report (1), but sorted alphabetically by golfer name.
(3) Per course, the same as report (1), but sorted in descending order of score on that course.

Each report contains one output record per golfer.

Worksheet

External input condition	Valid equivalence classes	Invalid equivalence classes

There are many possible solutions to this exercise. One possible solution can be found in Appendix D.

Boundary-value analysis
Boundary-value analysis is a variant and refinement of equivalence partitioning with two major differences:

First, rather than selecting any element in an equivalence class as being representative, elements are selected such that each edge of the EC is the subject of a test. Boundaries are always a good place to look for defects.

Second, rather than focusing exclusively on input conditions, output conditions are also explored by defining output ECs. What can be output? What are the classes of output? What should I create as an input to force a useful set of classes that represent the outputs that ought to be produced?

The guidelines for boundary-value analysis are:

- If an input specifies a range of valid values, write test cases for the ends of the range and invalid-input test cases for conditions just beyond the ends.
  Example: If the input requires a real number in the range 0.0 to 90.0 degrees, then write test cases for 0.0, 90.0, –0.001, and 90.001.
- If an input specifies a number of valid values, write test cases for the minimum and maximum number of values and one beneath and beyond these values.
  Example: If the input requires the titles of at least 3, but no more than 8, books, then write test cases for 2, 3, 8, and 9 books.
- Use the above guidelines for each output condition.

Boundary-value analysis is not as simple as it sounds, because boundary conditions may be subtle and difficult to identify. The method does not test combinations of input conditions.

Exercise on boundary-value analysis
Using the same functional design specification for GOLFSCORE described above, make a list of boundary values to be tested. Be sure to consider both input and output conditions. A possible solution can be found in Appendix D.

Error guessing
Error guessing is an *ad hoc* approach, based on intuition and experience, to identify tests that are considered likely to expose errors. The basic idea is to make a list of possible errors or error-prone situations and then develop tests based on the list. What are the most common error-prone situations we have seen before? Defects' histories are useful. There is a high probability that defects that have been there in the past are the kind that are going to be there in the future.

Some items to try are:

- empty or null lists/strings
- zero instances/occurrences
- blanks or null characters in strings
- negative numbers.

One of the studies done by Myers (1979) states that the probability of errors remaining in the program is proportional to the number of errors that have been found so far. This alone provides a rich source of focus for productive error guessing.

Cause-effect graphing
Cause-effect graphing is a systematic approach to selecting a high-yield set of test cases that explore combinations of input conditions. It is a rigorous method for transforming a natural-language specification into a formal-language specification, and exposes incompleteness and ambiguities in the specification.

Deriving cause-effect tests
Cause-effect tests are derived as follows:

- Decompose the specification into workable pieces.
- Identify causes and their effects.
- Create a (Boolean) cause-effect graph.
- Annotate the graph with constraints describing combinations of causes and/or effects that are impossible.
- Convert the graphs into a limited-entry decision table by methodically tracing state conditions in the graph. Each column in the table represents a test case.
- The columns in the decision table are converted into test cases.

Cause-effect graphing explores combinations of input conditions. It produces non-redundant, high-yield tests. It is useful in functional (external) specification verification, because it exposes errors in the specification. However, it is difficult and time-consuming to implement. It is a much more practical proposition when there is a tool to convert the graph into a decision table.

Syntax testing
Syntax testing is a systematic method of generating valid and invalid input data to a program. It is applicable to programs that have a hidden language that defines the data (e.g., the interactive commands to operating systems and subsystems). It is usually ineffective if the language is explicit and formally developed, as in compilers.

Syntax testing is basically a shotgun method that relies on creating many test cases, and is not useful for semantics testing (use function tests). The key to syntax testing is to learn how to recognize hidden languages.

The steps in syntax testing are:

- Identify the target language (explicit or implicit).
- Define the language syntax formally.

- Test the valid cases first by covering the definition graph.
- Design tests, level by level, top to bottom, making only one error at a time, one level at a time.
- Test the invalid cases.
- Automate the creation and execution of the tests.

State transition testing
State transition testing is an analytical method, using finite-state machines, to design tests for programs that have many similar, but slightly different, control functions. It is primarily a functional testing tool and also has a high payoff in functional design verification.

Graph matrix
A graph matrix is a simpler representation of a graph to organize the data. With a graph represented as a square matrix each row represents a node in the graph (node = 1, ..., n). Each column represents a node in the graph (node = 1, ..., n) and M(I,J) defines the relationship (if any) between node I and node J. (It is usually a sparse matrix because there is often no relationship between certain nodes.)

A graph matrix is used for proving things about graphs and for developing algorithms.

White-box methods for internals-based tests

Once white-box testing is started, there are a number of techniques to ensure the internal parts of the system are being adequately tested and that there is sufficient logic coverage.

The execution of a given test case against program P will exercise (cover) certain parts of P's internal logic. A measure of testedness for P is the degree of logic coverage produced by the collective set of test cases for P. White-box testing methods are used to increase logic coverage.

There are four basic forms of logic coverage:

(1) statement coverage
(2) decision (branch) coverage
(3) condition coverage
(4) path coverage.

White-box methods defined and compared
Figure 8.3 illustrates white-box methods. For example, to perform condition coverage, tests covering characteristics 1 and 3 are required. Tests covering 2 and 4 are not required. To perform multiple condition coverage, tests covering characteristics 1 and 4 are required. Such tests will automatically cover characteristics 1 and 2.

		Statement coverage	Decision coverage	Condition coverage	Decision/ condition coverage	Multiple condition coverage
1	Each statement is executed at least once	Y	Y	Y	Y	Y
2	Each decision takes on all possible outcomes at least once	N	Y	N	Y	implicit
3	Each condition in a decision takes on all possible outcomes at least once	N	N	Y	Y	implicit
4	All possible combinations of condition outcomes in each decision occur at least once	N	N	N	N	Y

Figure 8.3 The white-box methods defined and compared. Each column in this figure represents a distinct method of white-box testing, and each row (1–4) defines a different test characteristic. For a given method (column), "Y" in a given row means that the test characteristic is required for the method. "N" signifies no requirement. "Implicit" means the test characteristic is achieved implicitly by other requirements of the method. (© 1993, 1994 Software Development Technologies)

Exhaustive path coverage is generally impractical. However, there are practical methods, based on the other three basic forms, which provide increasing degrees of logic coverage.

Example of white-box coverage

To clarify the difference between these coverage methods, consider the following Pascal procedure. The goal of the example is to list one possible set of tests (sets of input data) which satisfies the criteria for each of the white-box coverage methods.

The liability procedure:

```
procedure liability (age, sex, married, premium ) ;
begin
    premium := 500 ;
    if ( (age < 25) and (sex = male) and (not married) ) then premium :=
    premium + 1500 ;
    else ( if ( married or (sex = female) ) then
                premium := premium – 200 ;
        if ( (age > 45) and (age < 65) ) then
                premium := premium – 100 ; )
end ;
```


The three input parameters are age (integer), sex (male or female), and married (true or false). Keep in mind the following:

- *Statement coverage*: Each statement is executed at least once.
- *Decision coverage*: Each statement is executed at least once; each decision takes on all possible outcomes at least once.
- *Condition coverage*: Each statement is executed at least once; each condition in a decision takes on all possible outcomes at least once.
- *Decision/condition coverage*: Each statement is executed at least once; each decision takes on all possible outcomes at least once; each condition in a decision takes on all possible outcomes at least once.
- *Multiple/condition coverage*: Each statement is executed at least once; all possible combinations of condition outcomes in each decision occur at least once.

A logic coverage methods solution for the liability (insurance) procedure follows. The following notation is used in each table shown below.

The first column of each row denotes the specific "IF" statement from the exercise program. For example, "IF-2" means the second IF statement in the sample program.

The last column indicates a test-case number in parentheses. For example, "(3)" indicates test-case number 3. Any information following the test-case number is the test data itself in abbreviated form. For example, "23 F T" means age = 23, sex = Female, and married = True.

An asterisk (*) in any box means "wild card" or "any valid input."

Statement coverage	Age	Sex	Married	Test case

There are only two statements in this program, and any combination of inputs will provide coverage for both statements.

Decision coverage	Age	Sex	Married	Test case
IF-1	< 25	Male	False	(1) 23 M F
IF-1	< 25	Female	False	(2) 23 F F
IF-2	*	Female	*	(2)
IF-2	>= 25	Male	False	(3) 50 M F
IF-3	<= 45	Female (n1)	*	(2)
IF-3	> 45, < 65	*	*	(3)

Note (n1): This input is not necessary for IF-3, but it is necessary to ensure that IF-1 is false [if age < 25 and married is false] so that the *else* clause of IF-1 (and hence IF-3) will be executed.

Condition coverage	Age	Sex	Married	Test case
IF-1	< 25	Female	False	(1) 23 F F
IF-1	>= 25	Male	True	(2) 30 M T
IF-2	*	Male	True	(2)
IF-2	*	Female	False	(1)
IF-3	<= 45	*	*	(1)
IF-3	> 45	*	*	(3) 70 F F
IF-3	< 65	*	*	(2)
IF-3	>= 65	*	*	(3)

Note: These test cases fail to execute the *then* clauses of IF-1 and IF-3 as well as the (empty) *else* clause of IF-2.

Decision/condition coverage	Age	Sex	Married	Test case
IF-1 (decision)	< 25	Male	False	(1) 23 M F
IF-1 (decision)	< 25	Female	False	(2) 23 F F
IF-1 (condition)	< 25	Female	False	(2)
IF-1 (condition)	>= 25	Male	True	(3) 70 M T
IF-2 (decision)	*	Female	*	(2)
IF-2 (decision)	>= 25	Male	False	(4) 50 M F
IF-2 (condition)	*	Male	True	(3)
IF-2 (condition)	*	Female	False	(2)
IF-3 (decision)	<= 45	*	*	(2)
IF-3 (decision)	> 45, < 65	*	*	(4)
IF-3 (condition)	<= 45	*	*	(2)
IF-3 (condition)	> 45	*	*	(4)
IF-3 (condition)	< 65	*	*	(4)
IF-3 (condition)	>= 65	*	*	(3)

Note: The above chart is simply all of the decisions (from the decision-coverage chart) merged with all of the conditions (from the condition-coverage chart) and then minimizing the number of test cases.

Multiple condition coverage	Age	Sex	Married	Test case
IF-1	< 25	Male	True	(1) 23 M T
IF-1	< 25	Male	False	(2) 23 M F
IF-1	< 25	Female	True	(3) 23 F T
IF-1	< 25	Female	False	(4) 23 F F
IF-1	>= 25	Male	True	(5) 30 M T
IF-1	>= 25	Male	False	(6) 70 M F
IF-1	>= 25	Female	True	(7) 50 F T
IF-1	>= 25	Female	False	(8) 30 F F
IF-2	*	Male	True	(5)
IF-2	*	Male	False	(6)
IF-2	*	Female	True	(7)
IF-2	*	Female	False	(8)
IF-3	<= 45, >= 65	*	*	impossible
IF-3	<= 45, < 65	*	*	(8)
IF-3	> 45, >= 65	*	*	(6)
IF-3	> 45, < 65	*	*	(7)

Validation activities

Validation activities can be divided into the following:

(1) Low-level testing
  (i) unit (module) testing
  (ii) integration testing.
(2) High-level testing
  (i) usability testing
  (ii) function testing
  (iii) system testing
  (vi) acceptance testing.

Low-level testing

Low-level testing involves testing individual program components, one at a time or in combination. It requires intimate knowledge of the program's internal structure and is therefore most appropriately performed by development.

Forms of low-level testing are:

- unit (module) testing
- integration testing.

Unit (module) testing

Unit or module testing is the process of testing the individual components (subprograms or procedures) of a program. The purpose is to discover discrepancies between the module's interface specification and its actual behavior.

Unit testing manages the combinations of testing. It facilitates error diagnosis and correction by development and it allows parallelism, in other words, testing multiple components simultaneously.

Testing a given module (X) in isolation may require:

(1) a driver module which transmits test cases in the form of input arguments to X and either prints or interprets the results produced by X;
(2) zero or more "stub" modules each of which simulates the function of a module called by X. It is required for each module that is directly subordinate to X in the execution hierarchy. If X is a terminal module (i.e., it calls no other modules), then no stubs are required.

Integration testing

Integration testing is the process of combining and testing multiple components together. The primary objective of integration testing is to discover errors in the interfaces between the components.

Integration testing can be performed on several levels. We can integrate and test the various modules of a program, the programs of a subsystem, the subsystems of a system, the systems of a network, and so on. There are a number of alternatives in integration testing.

In non-incremental "big bang" integration, all components are combined at once to form the program. The integrated result is then tested. While often used in practice, it is one of the least effective approaches. Debugging is difficult since an error can be associated with any component.

Incremental integration is where we unit test the next program component after combining it with the set of previously tested components. There are two approaches to incremental integration: bottom-up and top-down. Incremental integration has a number of advantages. It requires less work in the sense of fewer driver or stub modules. Errors involving mismatched component interfaces will be detected earlier. Debugging is easier because errors found are usually associated with the most recently added component. More thorough testing may result, because the testing of each new component can provide further function and logic coverage of previously integrated components.

The steps in bottom-up integration are:

- Begin with the terminal modules (those that do not call other modules) of the hierarchy.
- A driver module is produced for every module.
- The next module to be tested is any module whose subordinate modules (the modules it calls) have all been tested.
- After a module has been tested, its driver is replaced by an actual module (the next one to be tested) and its driver.

The steps in top-down integration are:

- Begin with the top module in the execution hierarchy.
- Stub modules are produced, and some may require multiple versions.
- Stubs are often more complicated than they first appear.
- The next module to be tested is any module with at least one previously tested superordinate (calling) module.
- After a module has been tested, one of its stubs is replaced by an actual module (the next one to be tested) and its required stubs.

Bottom-up integration has the disadvantage that the program as a whole does not exist until the last module is added. Top-down integration has the advantage that a skeletal version of the program can exist early and allows demonstrations. However, for top-down integration, the required stubs could be expensive. There is no clear winner here; the advantages of one are the disadvantages of the other, and the choice for testing is often based on the choice made for development.

Here again is an opportunity to take risk management into consideration. An effective alternative is to use a hybrid of bottom-up and top-down, prioritizing the integration of modules based on risk; e.g., the modules associated with the highest risk functions are integration tested earlier in the process than modules associated with low risk functions. Although there is a human tendency to want to do the easy things first, this hybrid approach advises just the opposite.

There are still other incremental integration strategies, which are usually variants of top-down and bottom-up (see References section at the end of the chapter for more details).

High-level testing

High-level testing involves testing whole, complete products. For purposes of objectivity, it is most appropriately performed outside the development organization, usually by an independent test group. Forms of high-level testing are:

- usability testing
- function testing
- system testing
- acceptance testing.

Usability testing

As computers become more affordable, software products are targeted at a larger user base. User expectations are steadily increasing and product purchases are increasingly based on usability. The goal is to adapt software to users' actual work styles, rather than forcing users to adapt their work styles to the software. Testing software for usability is an old idea whose time has come.

Usability testing involves having the users work with the product and observing their responses to it. Unlike Beta testing, which also involves the user, it should be done as early as possible in the development cycle. The real customer is involved as early as possible, even at the stage when only screens drawn on paper are available. The existence of the functional design specification is the prerequisite for starting.

Like all testing, it's important that our objectives are defined. How easy is it for users to bring up what they want? How easily can they navigate through the menus? We should try to do usability testing two or three times during the life cycle (see Figure 8.4).

Usability testing is the process of attempting to identify discrepancies between the user interfaces of a product and the human engineering requirements of its potential users. Testing of the user documentation is an essential component. Usability testing collects information on specific issues from the intended users. It often involves evaluation of a product's presentation rather than its functionality.

Historically, usability testing has been one of many components of system testing because it is requirements-based. Today, its importance and its pervasiveness throughout the development cycle have elevated it to a more prominent role.

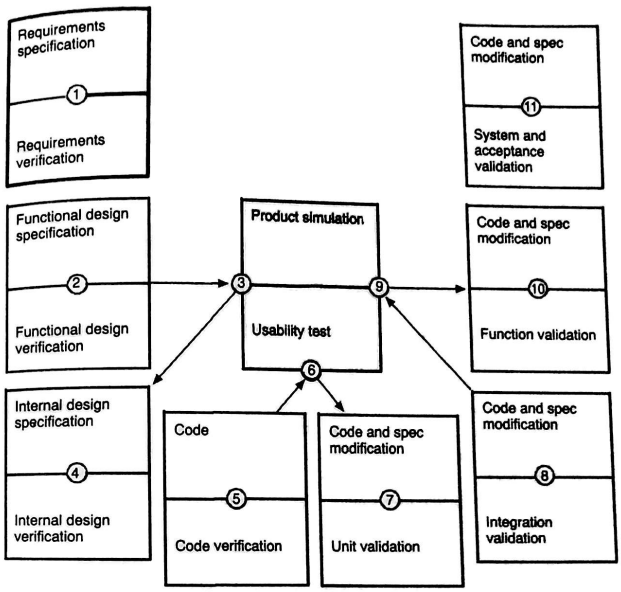


Figure 8.4 The SDT Dotted-U Model, usability test. (© 1993, 1994 Software Development Technologies)

Usability testing is considered a validation activity rather than a verification activity, because it requires a real user to interact with the end product by executing it in either a simulated or a real form. Usability characteristics which can be tested include the following:

- *Accessibility*: Can users enter, navigate, and exit with relative ease?
- *Responsiveness*: Can users do what they want, when they want, in a way that's clear?
- *Efficiency*: Can users do what they want in a minimum amount of steps and time?
- *Comprehensibility*: Do users understand the product structure, its help system, and the documentation?

The usability test process is as follows:

- Define the objectives of the test.
- Define the subjects precisely and recruit them.
- Plan the tests and develop all necessary materials.
- Put subjects in a workroom or lab with their workstations, possibly using video cameras, and a one-way window to an observation room.
- Conduct the test.
- Using video cameras and/or human observers, record the subject's every word and gesture.
- Experts and developers analyze results and recommend changes.

Types of usability tests include the following:

- *Freeform tasks*: Any unplanned task by a user.
- *Structured procedure scripts*: Pre-defined, written scripts containing step-by-step instructions for the user to follow.
- *Paper screens*: The researcher plays the role of computer for the user before any prototype or mock-up is available.
- *Prototype mock-ups*: Using a preliminary prototype rather than the final product.
- *Field trials in the user's office*: Using a prototype or the final product in the user's office.

Usability test methods can be obtrusive when a researcher, or an automated module in the product itself, guides the subject through a script and asks questions. They are unobtrusive when the user works alone or the usability professionals remain silent behind the one-way window.

Function testing

Function testing is the process of attempting to detect discrepancies between a program's functional specification and its actual behavior. When a discrepancy is detected, either the program or the specification could be incorrect. All black-box methods for function-based testing are applicable (see Figure 8.5).

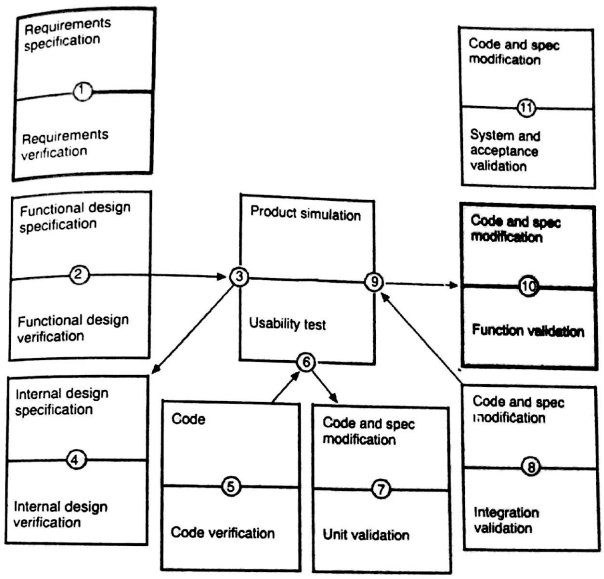


Figure 8.5 The SDT Dotted-U Model, function testing. (© 1993, 1994 Software Development Technologies)

Function testing is performed by a testing group before the product is made available to customers. It can begin whenever the product has sufficient functionality to execute some of the tests, or after unit and integration testing have been completed.

Function coverage is measured as follows:

The execution of a given test case against program P will exercise (cover) certain parts of P's external functionality. A measure of testedness for P is the degree of function coverage produced by the collective set of test cases for P. Function coverage can be measured with a function coverage matrix. Black-box testing methods are used to increase function coverage.

The steps of function testing are:

- Decompose and analyze the functional design specification.
- Partition the functionality into logical components and for each component, make a list of the detailed functions.
- For each function, use the analytical black-box methods to determine inputs and outputs.
- Develop the functional test cases.
- Develop a function coverage matrix.
- Execute the test cases and measure logic coverage.
- Develop additional functional tests, as indicated by the combined logic coverage of function and system testing.

A function coverage matrix (Figure 8.6) is simply a matrix or table listing specific functions to be tested, the priority for testing each function, and the test cases that contain tests for each function.

System testing

System testing is the most misunderstood and the most difficult testing activity. Despite its name, system testing is not the process of function testing the completely integrated system or program. Given function testing, this would be redundant. System testing is the process of attempting to demonstrate that a program or system does not meet its original requirements and objectives, as stated in the requirements specification.

System testing is difficult. There can be no design methodologies for test cases because requirements and objectives do not, and should not, describe the program's functions in precise terms. Requirements must be specific enough to be testable but general enough to allow freedom in the functional design. The specificity necessary for rigorous, universally applicable, technical methods is absent in a requirements specification (see Figure 8.7).

Because there is no methodology, system testing requires a great deal of creativity. We have to keep thinking from the perspective of the user, and the problem the user is trying to solve. System tests are designed by analyzing the requirements specification and then formulated by analyzing the functional design specification or user documentation. This is an ideal way to test user documentation, but it is often impractical because the manuals are usually not available when system test cases must be formulated.

Instead of using a special methodology, we can use various categories of system test cases. Requirements coverage can be stated as follows:

> The execution of a given test case against program P will address (cover) certain requirements of P. A measure of testedness for P is the degree of requirements coverage produced by the collective set of test cases for P.

Functions/Inputs	Priority	Test cases

Figure 8.6 Function coverage matrix form. (© 1993, 1994 Software Development Technologies)

Black-box testing methods are used to increase requirements coverage. Requirements coverage can be measured with a requirements coverage matrix or a requirements tracing matrix.

Types/goals of system testing are as follows:

- *Volume testing*: to determine whether the program can handle the required volumes of data, requests, etc.
- *Load/stress testing*: to identify peak load conditions at which the program will fail to handle required processing loads within required time spans.
- *Security testing*: to show that the program's security requirements can be subverted.
- *Usability (human factors) testing*: to identify those operations that will be difficult or inconvenient for users. Publications, facilities, and manual procedures are tested.
- *Performance testing*: to determine whether the program meets its performance requirements.
- *Resource usage testing*: to determine whether the program uses resources (memory, disk space, etc.) at levels which exceed requirements.
- *Configuration testing*: to determine whether the program operates properly when the software or hardware is configured in a required manner.

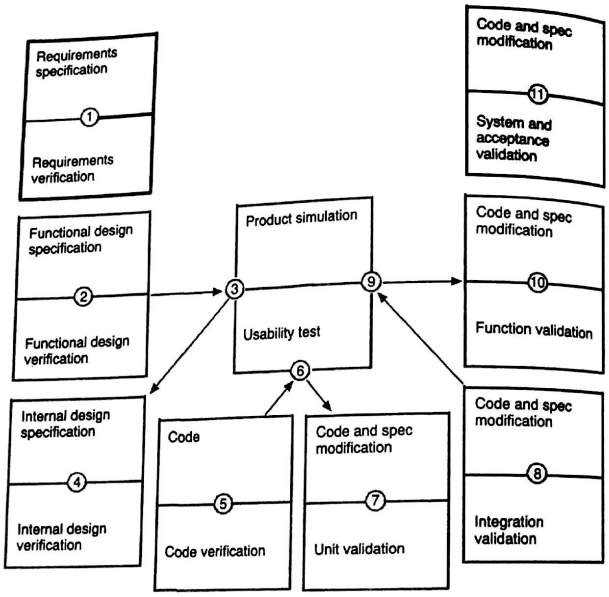


Figure 8.7 The SDT Dotted-U Model, system testing. (© 1993, 1994 Software Development Technologies)

- *Compatibility/conversion testing*: to determine whether the compatibility objectives of the program have been met and whether the conversion procedures work.
- *Installability testing*: to identify the ways in which the installation procedures lead to incorrect results.
- *Recovery testing*: to determine whether the system or program meets its requirements for recovery after a failure.
- *Serviceability testing*: to identify conditions whose serviceability needs will not meet requirements.
- *Reliability/availability testing*: to determine whether the system meets its reliability and availability requirements.

System testing is performed by a testing group before the product is made available to customers. It can begin whenever the product has sufficient

functionality to execute some of the tests or after unit and integration testing are completed. It can be conducted in parallel with function testing. Because the tests usually depend on functional interfaces, it may be wise to delay system testing until function testing has demonstrated some pre-defined level of reliability, e.g., 40% of the function testing is complete.

The steps of system testing are:

- Decompose and analyze the requirements specification.
- Partition the requirements into logical categories and, for each component, make a list of the detailed requirements.
- For each type of system testing:
  - For each relevant requirement, determine inputs and outputs.
  - Develop the requirements test cases.
- Develop a requirements coverage matrix which is simply a table in which an entry describes a specific subtest that adds value to the requirements coverage, the priority of that subtest, the specific test cases in which that subtest appears.
- Execute the test cases and measure logic coverage.
- Develop additional tests, as indicated by the combined coverage information.

Acceptance testing

Acceptance testing is the process of comparing the end product to the current needs of its end users. It is usually performed by the customer or end user after the testing group has satisfactorily completed usability, function, and system testing. It usually involves running and operating the software in production mode for a pre-specified period (see Figure 8.8).

If the software is developed under contract, acceptance testing is performed by the contracting customer. Acceptance criteria are defined in the contract. If the product is not developed under contract, the developing organization can arrange for alternative forms of acceptance testing – ALPHA and BETA.

ALPHA and BETA testing are each employed as a form of acceptance testing. Often both are used, in which case BETA follows ALPHA. Both involve running and operating the software in production mode for a pre-specified period. The ALPHA test is usually performed by end users inside the developing company but outside the development organization. The BETA test is usually performed by a selected subset of actual customers outside the company, before the software is made available to all customers.

The first step in implementing ALPHA and BETA is to define the primary objective of the test: *progressive testing*, and/or *regressive testing*. Progressive testing is the process of testing new code to determine whether it contains errors. Regressive testing is the process of testing a program to determine whether a change has introduced errors (regressions) in the unchanged code.

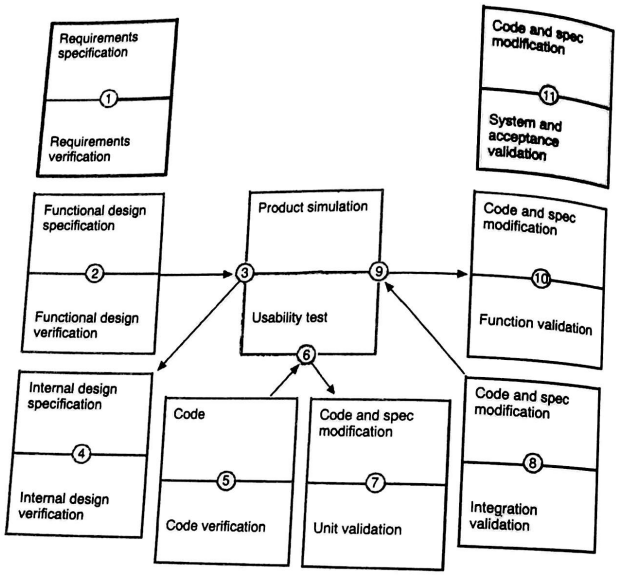


Figure 8.8 The SDT Dotted-U Model, acceptance testing. (© 1993, 1994 Software Development Technologies)

Both ALPHA and BETA are often more effective as regressive, rather than progressive, tests. Both ALPHA and BETA require careful selection of users. Users should be motivated to use the product. Users provide good coverage of hardware and software configurations. Site location is important for support purposes.

You should give careful consideration to the duration of the test. It must be long enough to get the desired feedback. The duration depends on whether or not the new software has an associated learning curve or requires new development by users.

Bilateral plans and agreements are recommended. The user agrees to the duration of the test period and to specified amounts and types of usage during the period and agrees to report problems, progress, and status in a timely fashion. The provider agrees to special support arrangements. Some organizations offer a financial bonus on finding defects of high severity.

Retesting

Why retest? Because any software product that is actively used and supported must be changed from time to time, and every new version of a product should be retested.

Progressive testing and regressive testing

Most test cases, unless they are truly throw-away, begin as progressive test cases and eventually become regression test cases for the life of the product. Regression testing is not another testing activity. It is a re-execution of some or all of the tests developed for a specific testing activity. It may be performed for each activity (e.g., unit test, usability test, function test, system test, etc.).

New versions of a product submitted for testing will be one of the following:

(1) The initial input to a new round of testing, ultimately resulting in a major new version of the product.
(2) A subsequent update to the initial input, regardless of whether it has been made available to customers (i.e., has satisfied its release criteria), or not.

In theory, a subsequent update requires just as much retesting as the initial input, although both of these can introduce serious regressions. Treating them differently involves risk. This risk can be reduced by managing change carefully with configuration management, and understanding the effects of a given change on the product.

Two testing policies at opposite extremes are:

(1) The testing organization, in the spirit of teamwork, agrees to test preliminary and perhaps incomplete versions of a product that are not candidates for general release. Here testing procedures are typically informal. Formal retesting will usually be required when the product is declared a candidate for release. Such a policy could be the target of much abuse of the test organization, which spends all of its time in this mode instead of building an automated regression test library.
(2) The testing organization will accept for testing only those versions of a product that are identified as candidates for release to customers. This is a bit heavy-handed, but sometimes necessary.

In some organizations, the testing function will not accept a product for testing until that product has successfully passed a pre-defined set of acceptance tests. Such acceptance tests are carefully selected to ensure that subsequent testing activities will be productive. This means that the product's basic functions are sufficiently sound so that there is a good probability that all other tests can be executed without major obstructions.

While it may be very helpful to the product development team, executing tests against an incomplete product will usually require the skills and expertise of the test developers. If test developers spend all of their time executing tests, there is no time left for developing more tests. It is important to consider all aspects, including the schedules and economics of the situation, before committing to test products that are not candidates for release. Many testing organizations have rigid criteria for entry of a product into testing (i.e., criteria for acceptance by the testing organization).

Designing for test execution

When designing test cases intended to detect errors, there are test-execution considerations to keep in mind:

- many small test cases vs. a few large ones
- dependencies among test cases
- host environment for test execution
- testpoints.

A test case contains one or more subtests and the objective of every subtest is to detect errors. A subtest might be passing to the product under test:

- valid inputs exclusively
- invalid inputs exclusively
- a combination of valid and invalid inputs.

When a test case causes a failure, a number of things could occur, depending on the test case itself, the test environment, and the product under test. The execution of the test case can be prematurely terminated, in which case the subsequent subtests are not executed. This means that the subsequent subtests can't be executed until the error is corrected or the test case is modified. Alternatively, the execution of the test case can continue as though no failure had occurred.

Test conditions can be masked on several levels:

(1) *Inter-test case:* When one test case (X) depends on another (Y), then when Y cannot satisfy X's dependency because Y caused a failure, the remainder of X may not be executed.

(2) *Intra-test case, inter-subtest:* Within a single test case, when one subtest causes a failure, the remaining subtests may not be executed.

(3) *Intra-subtest, inter-input:* Within a single subtest, when one input causes a failure, the remaining inputs may not be tested.

Example of masking test conditions

- In testing a programmatic interface, a subtest invokes one of the product's procedures (XYZ):
  CALL XYZ (invalid, valid, invalid, valid, invalid);
- XYZ will check the validity of its input parameters in some order that is usually unknown to the tester.
- When the first invalid parameter is detected, XYZ will terminate execution prematurely by either aborting or returning an error code to the caller, leaving all other invalid parameters unchecked.

Note that the above example permanently masks input conditions. Even a valid input, when it creates a failure in the product being tested, can temporarily mask other input conditions until the error is corrected.

Guidelines for test execution

- Use large test cases for subtests using valid inputs.
- Use one test case per subtest using one invalid input.
- A test case can depend on another program but should not depend on another test case.
- The state of the host system should be identical when the execution of each test case is initiated.
- New files should not accumulate.
- Operating modes should not change. This guideline is related to the repeatability of each test case. When a test case causes a failure, it is often difficult to diagnose the cause. The cause can be even further obscured by allowing the underlying environment to get out of control. This guideline is also applicable at the subtest level within a test case.

Testpoints

It is sometimes difficult for test cases to create the conditions necessary for comprehensively testing a product. To overcome these obstacles, testing may ask development to add testpoints to the product. A testpoint is a permanent point in the product which:

- interrogates the value of a variable which can be set via external means by the tester, either manually or programmatically;
- performs one or more specific actions, as indicated by the value of the variable;
- does nothing if the variable is set to its default value.

Testpoints are used to halt the system to test recovery procedures, to introduce timing delays, to invoke a procedure supplied by the tester, and to generate an input/output error condition in a channel, controller, or device to test recovery procedures.

Recommended strategy for validation testing

- Methodically identify, design, and develop function-based tests for function testing, and using black-box methods and requirements-based tests for usability and system testing.
- Run the tests and, using an automated tool, measure their collective internal logic coverage.
- Using white-box methods, methodically identify and develop supplementary internals-based tests as necessary to improve internal logic coverage.
- Review and analyze test results.
- Intensify the testing of any area that exhibits a disproportionately high number of errors.

Externals-based testing should be undertaken by people who have no knowledge of the internal structure of the program to be tested. Such knowledge would bias the tests and can be avoided either by using different people to design the different types of tests, or by using the same people but designing the internals-based tests after the externals-based tests.

- Key black-box methods for function-based testing
  - Equivalence partitioning
  - Boundary-value analysis
  - Error guessing
- White-box methods for internals-based testing
  - Statement coverage
  - Decision coverage
  - Decision/condition coverage
  - Multiple condition coverage

Figure 8.9 Methodology summary. (© 1993, 1994 Software Development Technologies)

Type of testing	Performed by
• Low-level testing	
– Unit (module) testing	Development
– Integration testing	Development
• High-level testing	
– Usability testing	Independent test organization
– Function testing	Independent test organization
– System testing	Independent test organization
– Acceptance testing	Customers

Figure 8.10 Activity summary. (© 1993, 1994 Software Development Technologies)

References

Sources of more detailed information on the analytical methods of black-box and white-box testing and for more detailed information on unit testing, integration testing, and system testing:

Beizer, B. (1984). *Software System Testing and Quality Assurance.* Van Nostrand Rheinhold.
Beizer, B. (1990). *Software Testing Techniques.* Van Nostrand Rheinhold.
Myers, G.J. (1976). *Software Reliability: Principles and Practices.* John Wiley.
Myers, G.J. (1979). *The Art of Software Testing.* John Wiley.

For additional information on usability testing, see:

Myers, G.J., Brad, A. and Rosson, Mary Beth (1993). "Survey on User Interface Programming," *CHI '92 Conference Proceedings.* ACM Conference on Human Factors in Computing Systems.
Rosenbaum, Stephanie (1993). "Alternative Methods for Usability Testing," *Software Testing, Analysis, & Review (STAR) Conference Proceedings.*

Refer also to:

- SIGCHI (ACM Special Interest Group on Human Factors in Computing Systems)
- Usability Professionals Association

Organizations specializing in usability testing:

- American Insitutes for Research, Palo Alto, CA
- IBM User Interface Institute
- Mead Data Central, Inc, Miamisburg, OH
- Multimedia Research, Bellport, NY
- Usability Sciences Corp., Dallas, TX

Chapter 9
Controlling validation costs

If we had unlimited resources we could do all the testing we wanted. In the real world of software projects, we are almost always short of time or money – or both. The discomforting side of it is that not only do we not know how to reduce the costs on a systematic basis, but also we often don't know what the real cost of our testing is in the first place.

Discovering what the real costs are is only possible through measurement (see Chapter 12). But how do we get the costs we have – particularly the validation costs – under the kind of control we are aiming to establish for our testing practices in general? This chapter addresses the main considerations for building cost control into the overall testing strategy.

An important element is testware. Testware is the collection of major work products (deliverables) of testing. The primary objective of testware is to maximize the testing yield, by maximizing the potential for detecting errors and minimizing the number of tests required.

There are additional objectives of testware unrelated to error detection as such, that are extremely important to cost control. These objectives are to minimize the cost of performing tests, the cost of test maintenance, and the cost of test development.

Minimizing the cost of performing tests

The one-time costs of performing tests as progressive tests are usually not very important. The recurring cost of performing these same tests as regression tests for each new version of the product throughout its lifetime is important in proportion to expected need – and that's usually very important.

Cost components in performing tests include:

- pre-run setup cost
- execution cost
- post-run cost.

Pre-run setup costs

Here we are concerned about minimizing the amount of time, the amount of labor, and in particular the amount of skilled labor required to do various essential tasks. These include configuring the hardware, configuring the software, establishing the test environment (restoring files and initialization), and identifying the tests to be run.

Execution costs

Here we aim to minimize the total execution time and dedicated equipment required. Execution time includes attended time (any part of the test execution that requires a manual action by the user or operator where we need to minimize time, labor and skills) plus unattended time.

What are we going to do when we have to re-run our tests? Are we going to run them all each time? These choices are the main engine for cost control in this area, and the decision is based on risk versus cost considerations.

Full regression testing (running all tests) minimizes the risk but increases the cost of test execution. Partial regression testing (running a selected subset of tests) reduces the cost of test execution but increases the risk.

The cost of selecting the right tests for partial regression testing could be high and should be weighed against the reduced cost of test execution. This is dependent on the particular environment. If it is highly automated, sometimes it is cost effective to run all the tests again.

There are two ways to select tests for partial regression testing:

(1) identify and select all tests that would retest everything affected by the change, or

(2) identify and select tests that would retest only those things closely and directly related to the change, knowing that the selection process is less than perfect.

Note that there is less risk associated with (1). Also note that (2) is very subjective, and may still require a significant amount of product knowledge. The general recommendation is to use partial regression testing only if full regression testing is prohibitive.

Post-run costs

Here we aim to minimize the amount of time and the amount of skilled labor required for the analysis and documentation of test results, and for tear-down of the test environment and the restoration of the previous environment.

Recommendations for minimizing the cost of performing tests

Reconfiguring hardware can be time consuming and expensive. When it comes to pre-run, consider the use of dedicated hardware that is permanently configured for the test environment. Automate the configuration of the software and the test environment as much as possible, and, to the extent possible, automate the process of identifying and selecting the tests to be run.

Test execution should be automated as far as possible so as to require no manual assistance. If some portion of the test execution must be attended, it's better to use a junior technician than the test developer. On the other hand, a test developer, repeatedly faced with the task of attending test execution, will usually discover a way to automate it!

When it comes to post-run, to the extent possible, the comparison of test results with expected results should be automated to minimize analysis cost. There are two basic methods for automating tests:

(1) use a testing tool (using an existing tool is almost always more cost effective than building one);
(2) build the automation into the test case itself.

The two recommended candidates for automation are:

(1) manual interactions (by a user or by the system operator) required during execution;
(2) checking of test results.

Both methods apply to both candidates.

Using tools to automate tests

We can use capture/replay tools to automate the execution of tests. In post-run we should automate as much as possible. Taking three days, after the tests have all been run, to manually go through test results before we know whether the software has passed or failed the tests makes little sense given the time pressure that usually prevails at this point. Automation at this stage also eliminates human error.

To automate the checking of test results, use capture/replay or a script-driven simulator during execution and a comparator after execution. There is further and more general material on testing tools and specific information on capture/playback tools in Chapter 11.

Automating manual interactions

Automating manual interaction is desirable for the testing of any software. It is imperative for testing products with highly interactive user interfaces, for simultaneous usage of multi-user software, and for software with graphic user interfaces (GUIs).

The basic categories of manual actions are operator actions, which advance the execution of the tests, such as mounting a tape, initiating or resuming execution, and user actions required by interactive, script-driven testing.

Automating checking of test results is desirable for all tests because, in addition to eliminating manual work, it reinforces the need to pre-define the expected results. Depending on test case design, test results can be checked in two ways:

(1) During execution (on the fly) – an exception message is printed when actual and expected results disagree.
(2) After execution – actual test results are written to a file during execution and compared with expected results after execution.

Minimizing the cost of maintaining the tests

Regression tests should be faithfully maintained for the life of the software product. Like the software components for the product itself, all testware components should be placed under the control of a configuration management system.

The key maintenance tasks are:

- add a test for each confirmed problem reported;
- add progressive tests to test new changes;
- periodically review all test cases for continued effectiveness.

How do we determine the continued effectiveness of tests?

(1) *Each test case should be executable*; that is, the product being tested should not have introduced functional changes which prevent the intended execution of the test case.
(2) *Requirements-based and function-based tests should be valid*; that is, a change in product requirements or functionality should not have rendered the test case misleading (detecting errors that don't exist) or useless (failing to detect errors that it used to detect).
(3) *Each test case should continue to add value*; that is, it should not be completely redundant with another test case, but it may often be more cost effective to continue using it than to determine its added value.

To preserve their customers' investment in their products, many software producers commit to providing upward compatible functionality in each new release. This commitment can be exploited by the tester by maximizing the number of tests that use only those interfaces described in the functional specification. This applies to all tests, even requirements-based and internals-based tests. Such tests should serve as useful, maintenance-free regression tests for all releases of the product.

For usability reasons, software producers often make significant, incompatible changes in user interfaces of products from one release to the next. Such changes are usually in the interactive, as opposed to programmatic, interfaces of products. Such changes may be good for users, even considering the (usually small) cost of retraining.

However, these changes create more work (but provide job security) for testers. New tests must be developed. Old tests must be examined and affected tests must be changed or declared obsolete. Where there is a clear one-for-one change in a product interface, a good tool or a one-off conversion program might be able to modify the corresponding tests automatically.

There is a dark side to white-box testing. A test case that references a product interface for which there is no (expressed or implied) commitment to future compatibility is a high maintenance test case – invoking internal procedures directly, and/or examining or setting an internal data item. The number of such test cases should be minimized because they must be re-examined for continued validity prior to each use. Use this kind of testing carefully, but use it when it is needed. At the same time, try to get commitments to maintain compatibility for the future.

Axioms for test maintenance

The axioms for test maintenance are:

- Never alter a program to make the testing of that program easier (e.g., testing hooks), unless it is a permanent change.
- If a test case must reference an internal item in a product, always try to extract a commitment from development to make that item permanent.

Minimizing validation testware development costs

For validation testing, testware includes test cases and test data as well as supporting documentation such as test plans, test specifications, test procedures, and test reports.

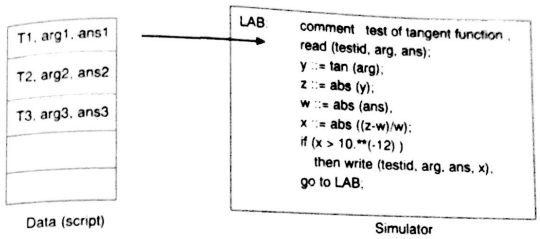


Figure 9.1 Example of a script-driven simulation test. Real (floating point) numbers will seldom be exactly equal. So the test for equality is described in the algorithm (asking whether their first 12 decimal digits are equal). (© 1993, 1994 Software Development Technologies)

The development of quality testware requires the same disciplines as are applied to good software engineering in general. We have to plan each stage, and understand what is required before the design begins. We need a detailed design before coding begins and we must comply with established coding standards. We should use inspections and walkthroughs on testware, as we would on any other important software product. Test users will need user documentation. Overall, testware should be treated as the important business asset that it is by placing it under control of a configuration management system.

The basic testware acquisition strategies are, quite simply, build, buy or reuse. Testware should always be reused if possible, which is why the maintenance of testware is so important.

Buy testware if it is available. For some standards-based software, appropriate testware can be purchased fairly easily (e.g., validation tests for C, UNIX, COBOL, Pascal). Sometimes useful test libraries can be bought from companies doing the same kind of work. Build testware only if absolutely necessary, because it is usually the least cost-effective solution. It can be built in-house, or by an outside contractor. Another important consideration in developing testware is the cost of automating execution versus the cost of executing non-automated tests, and the use of existing test tools.

Some types of software products or product-components lend themselves to testing via a script-driven simulator. Actual tests and their expected results are written and stored as data records (scripts) in a text file. One program (a simulator) is written, and it reads the data for each test, interprets and executes it, and compares the actual to the expected results.

Writing tests as data rather than as code can reduce development (and maintenance) costs. The larger the number of tests, the greater the savings. This is most useful when testing programmatic interfaces (see Figure 9.1).

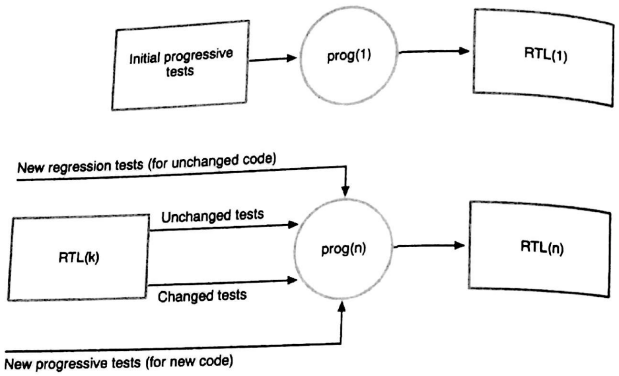


Figure 9.2 Regression test library evolution. (© 1993, 1994 Software Development Technologies)

The testware library

Because testware must be accessed frequently, test cases should be organized and cataloged in the form of a library for convenient access. Over the life of the testware, such a library affords significant cost savings in performing tests and maintaining them. An easy-to-use testware library will be used to advantage by others, not just its creators. A library should be organized in such a way that all test cases or any subset can be selected and executed with minimal effort (see Figure 9.2).

Define a structure and naming conventions that are most appropriate for the organization's particular testing needs. To make the library useful to others, collect related test cases that are normally run as a unit into (named) groups. A test library may contain many groups, and any given test case belongs to one or more groups.

The library structure and/or grouping and naming conventions should explicitly identify and differentiate:

- test cases and groups that require the same execution environment;
- high-maintenance tests that must be examined prior to each use;
- the nature (category, purpose, test-basis) of the test case or group.

The test library should contain:

- an index or roadmap;
- a description of the library structure and grouping and naming conventions;
- for each test group:
  - a description of the group and grouping criteria
  - the names of the test cases in the group
  - test procedure documentation (for the group)
  - a command file to execute all test cases in the group;
- for each test case:
  - test case and test procedure documentation
  - all associated files (source, object, data, etc), including a command file to execute the test case and command files to create all derived files.

Recommendations

- Review how accurate your information is on your present test spending.
- Evaluate the use of dedicated hardware that is permanently configured for the test environment.
- Automate the comparison of test results with expected results to minimize analysis cost.
- Investigate using capture/replay tools to automate test execution.
- Place all testware components under the control of a configuration management system.
- Plan a periodical review of all test cases for continued effectiveness.

Chapter 10
Testing tasks, deliverables, and chronology

Effective testing is planned. Successful testing requires a methodical approach involving discipline, structure, analysis, and measurement. Unorganized, shotgun, random approaches will almost always fail.

Each testing activity has one or more inputs (development or documentation deliverables) and consists of one or more tasks. Each task produces one or more outputs (testing deliverables).

This chapter describes the specific tasks and the deliverables associated with each testing activity, and the standards which provide help with these tasks and deliverables. It includes an outline for the content of each deliverable document, and where applicable, a reference to the IEEE/ANSI standard for the document. These tasks and deliverables are mapped into specific phases of the software life cycle, showing their relative timing and overlap.

Time and resources almost always limit the testing we can do in practice. At every level of testing (activity, task, subtask, etc.) it is important to set priorities on every aspect of the work being considered. The main basis for setting these priorities is risk.

Master test planning

An overview of testing tasks is as follows:

(1) Master test planning task
(2) Verification testing tasks (per activity)
  (i) planning
  (ii) execution
(3) Validation testing tasks (per activity)
  (i) planning
  (ii) testware development
  (iii) test execution
  (iv) testware maintenance.

The master test planning risk management considerations include:

- size and complexity of the product to be tested;
- criticality of the product. Critical software is software in which a failure could have an impact on safety or could cause large financial or social losses (IEEE/ANSI, 1986 [Std 1012-1986]);
- (SEI) development process maturity level;
- form of testing (full, partial, endgame, audit);
- staffing, experience, and organization.

(See Chapter 5 for more detail on the background to these considerations.)

A three-level priority scheme (LO, MED, HI) is usually sufficient. When the work is being performed the HI priority items are done first, then MED, and then, if time permits, LO.

In master test planning the aim is to get the big picture clear in a high-level document – a master schedule, master resource usage, a master life cycle, and quality assurance issues to be addressed. What kind of testing will we be doing? How much verification? What kind of validation? Do we do accept-ance testing? What overall strategy will we need?

One such "big picture" deliverable is the software verification and validation plan. Performing software verification and validation, as defined in the IEEE/ANSI Standard 1012-1986, provides for a comprehensive evaluation throughout each phase to help ensure that errors are detected and corrected as early as possible in the software life cycle, and project risk, cost, and schedule effects are lessened. Software quality and reliability are enhanced, proposed changes and their consequences can be quickly assessed, and transparency in the software process is improved.

Deliverable: software V&V plan (outline)
(IEEE/ANSI, 1986 [Std 1012-1986])

- Purpose
- Referenced documents
- Definitions
- Verification and validation overview
  – organization, master schedule, resources summary, responsibilities, tools, techniques, and methodologies
- Life-cycle verification and validation
  – tasks, inputs, and outputs per life-cycle phase
- Software verification and validation reporting
  – describes content, format, and timing of all V&V-reports
- Verification and validation administrative procedures
  – policies, procedures, standards, practices, conventions

IEEE Standard 1012-1986 contains charts and explanatory information that assist the author in preparing the plan. The overall intent of the plan is to describe the V&V activities that will be applied to the project. The Standard document defines the outline of the plan and basically serves as a checklist for the content of the plan. The software V&V plan is included in the list of minimum documentation requirements for critical software, and it is subordinate to the umbrella software quality assurance plan (SQAP).

The SQAP is the highest level, testing related document. It references the fact that there is a software V&V plan. The software QA plan is directed toward the development and maintenance of critical software; a subset of the standard may be applied to non-critical software. The purpose of the SQAP is to show the user, the developer, and the public the specific measures that are being taken to ensure quality in the software. It includes items such as how configuration management and relationships with suppliers and contractors will be handled. The SQAP is prepared by a SQA group or an appropriate representative body.

Deliverable: software quality assurance plan (outline)

(IEEE/ANSI, 1989 [Std 730-1989])

- Purpose
- Referenced documents
- Management
- Documentation
- Standards, practices, conventions, and metrics
- Life-cycle verification and validation
- Reviews and audits
- Test
- Problem reporting and corrective action
- Tools, techniques, and methodologies
- Code control
- Media control
- Supplier control
- Records collection, maintenance, and retention
- Training
- Risk management

Verification testing tasks and deliverables

An overview of verification activities is as follows:

(1) Activities
  (i) requirements verification
  (ii) functional design verification
  (iii) internal design verification
  (iv) code verification

(2) Verification tasks (per activity) include:
  (i) planning
  (ii) execution.

Verification test planning

Are we going to do requirements verification? What methods are we going to use: full-blown formal inspections, or walkthroughs, or desk checks? What areas of the work product will be verified? One hundred per cent of the requirements specified? Fifty per cent of the code? What are the risks involved in not doing it? What is the resource schedule and what are the responsibilities?

Verification test planning considerations include:

- the verification activity to be performed;
- the methods to be used (inspections, walkthroughs, etc.);
- the specific areas of the work product that will and will not be verified;
- the risks associated with any areas of the work product that will not be verified;
- prioritizing the areas of the work product to be verified;
- resources, schedule, facilities, tools, responsibilities.

The deliverables comprise one verification test plan per verification activity. For each verification activity to be performed (i.e., requirements verification, functional design verification, internal design verification, code verification), a verification test plan is produced. Its purpose is to describe in detail how the verification will be performed, the areas of the work product that will and will not be verified, associated risks and priorities, and other standard planning information. There is no IEEE/ANSI standard for this document.

Deliverable: verification test plan (outline)

- Test-plan identifier
- Introduction
- Verification activity (requirements vs. ...)
- Areas to be verified, not verified
- Tasks
- Responsibilities
- Staffing and training needs
- Schedule
- Risks and contingencies
- Approvals

Verification execution

Tasks:

- inspections, walkthroughs, technical reviews

Deliverables:

- inspection report (one per inspection)
- verification test report (one per verification activity)

Each individual inspection delivers a report on its activity. What was inspected? Who was there? How much did people work in advance? What was the error rate? Where were the defects found and in what category of severity? What conclusions did we come to about the product? Do we need to reinspect after the errors have been fixed? Was the whole meeting aborted? Why? How much rework is needed?

Deliverable: inspection report (outline)

- Inspection report identifier
- Test item and version
- Number of participants
- Size of the materials inspected
- Total preparation time of the inspection team
- Disposition of the software element
- Estimate of the rework effort and rework completion date
- Defect list
- Defect summary (number of defects by category)

The inspection report outline was derived from section 6.8 of IEEE/ANSI Standard for Software Reviews and Audits (Std 1028-1988), which sets minimum requirements for report content.

There is one more verification execution-related deliverable, the verification test report. This test report is a summary of verification activities. We aimed to verify 100% of the software related documentation, but how much did we end up actually verifying? What kind of internal issues came up that need resolving? This is what was achieved, this is what was not. It can be seen as an executive summary, which can be used to raise management awareness of the testing process, and draw their attention to issues that need to be addressed.

Deliverable: verification test report (outline)

- Verification report identifier
- Type of verification
- Test item and version
- Summary
- Variances (from requirements at start of phase)
- Internal issues (as a stand-alone entity)
- Log of verification steps (meetings, inspections, special actions, etc.)
- Summary of criticality and risk assessment
- List of open action items

There is one verification test report per work product (not one per inspection meeting). Its purpose is to summarize all the verification steps (tasks) that were performed for this verification activity. There is no IEEE/ANSI standard for this document.

Validation testing tasks and deliverables

A summary of validation activities is as follows:

- Unit testing (by development)
- Usability testing
- Function testing
- System testing
- Acceptance testing

Validation tasks:

- High-level planning for all validation activities as a whole
- Testware architectural design
- Per activity:
  - detailed planning
  - testware development
  - test execution
  - test evaluation
  - testware maintenance

Validation test planning

What kind of test methods are we going to use? What kind of test automation, including tools? What kind of budget do we have? What support software and training do we need? How are we going to do our configuration management?

Validation test planning considerations:

- test methods
- facilities (for testware development vs. test execution)
- test automation
- support software (shared by development and test)
- configuration management
- risks (budget, resources, schedule, training, etc.)

These considerations apply to both the overall testing effort and each individual validation activity (in more detail).

One master validation test plan should be produced for the overall validation testing effort. One or more detailed validation test plans should be produced for each validation activity (unit testing, integration testing, usability testing, function testing, system testing, and acceptance testing).

The purpose of the master validation test plan is to provide an overview of the entire validation testing effort. It should identify the specific validation activities to be performed, the approximate resources required by each, a rough schedule for each, the overall training needs and the risks.

Deliverable: master validation test plan

(IEEE/ANSI, 1983 [Std 829-1983])

Purpose:

- To prescribe the scope, approach, resources, and schedule of the testing activities.

Outline:

- Test-plan identifier
- Introduction
- Test items
- Features to be tested
- Features not to be tested
- Approach
- Item pass/fail criteria
- Suspension criteria and resumption requirements
- Test deliverables
- Testing tasks
- Environmental needs
- Staffing and training needs
- Schedule
- Risks and contingencies
- Approvals

There is at least one detailed (validation test) plan per validation testing activity. The purpose of the detailed plan is to describe in detail how that validation activity will be performed:

- unit testing (by development)
- integration testing (by development)
- usability testing
- function testing
- system testing
- acceptance testing.

The test plan outline above may be used for both the master and detailed validation test plans. Every item in the outline is applicable to the detailed test plan where we itemize what we are going to test, the features to be tested, and the features not to be tested. In other words, for the detailed test plan, every item can still apply, but on a lower, more detailed level.

Test architecture design

An important supplement to the master validation test plan is the test architecture design. Architecture design is the process of designing a meaningful and useful structure to the tests as a whole. How do we organize the tests? Are they requirements-based tests, function-based tests, or internals-based tests? How do we categorize them? What is the logical grouping of tests we intend to execute together? How do we structure the test repository?

There is one and only one test architecture specification per major software product. It can be seen as the root document (roadmap) of the entire test repository.

The key architecture design considerations include:

(1) organization of the tests with respect to test basis (requirements- vs. function- vs. internals-based tests);
(2) categorization of the tests and grouping conventions;
(3) structure and naming conventions for the test repository;
(4) grouping of tests for reasonable execution periods.

Deliverable: test-architecture specification

- Test-specification identifier
- Introduction
- Test repository location
- Test repository storage and access conventions
- Test repository structure/organization
- Standards
- Grouping and naming conventions for all files

Testware development – detailed design and implementation

Testware is designed with the following objectives:

- detect as many errors as possible
- minimize test development costs
- minimize test execution costs
- minimize test maintenance cost.

Testware is software, so good software design and software engineering techniques apply to testware development.

The tasks of testware development include:

- detailed design
- implementation.

At this stage we are working towards the test design specification. What are we going to test? What are the priorities assigned to it? How are we going to put together the high-level test designs for groups of related items? How are we going to approach actually doing the tests?

Detailed design is the process of specifying the details of the test approach for a software feature or combination of features and identifying the associated test cases. Detailed design considerations are:

- satisfying test development objectives;
- conforming to the test architecture;
- design of each test case.

The deliverables of detailed design are test design specifications and test case specifications.

The basic steps of detailed test design

- identifying the items that should be tested;
- assigning priorities to these items, based on risk;
- developing high-level test designs for groups of related test items;
- designing individual test cases from the high-level designs.

Test item identification is the process of identifying all target items to be tested. The first step of test item identification is a careful study, decomposition, and analysis of the requirements and functional design specifications. Using black-box methodologies, a list of target test items for function-based tests is developed. A test item is the subject of a subtest. Using experience and ingenuity, we create a separate list of target test items for requirements-based tests for each relevant type of system testing. In some test literature, test items are called test objectives, and the collection of test items is called an inventory.

Anything at our disposal can be used to develop separate lists of test items for requirements-based tests for each type of system test. Any or all of the following types might apply, and may require special approaches to test design, special tools or hardware or environment. For each type of system testing, first consider whether the type is applicable. If it is, develop a list of test items based on the specific requirements for that type of system test.

Types of system tests (for requirements-based tests):

- volume
- usability
- performance
- configuration
- compatibility/conversion
- reliability/availability
- load/stress
- security
- resource usage
- installability
- recovery
- serviceability.

Once again, we can't do everything, but we have to be sure we have done the essentials, and we need to assign priorities (based on risk) to our lists of target test items. Steps for refining the lists of target test items include:

(1) If the lists were developed independently, compare the two lists, and eliminate redundant test items.
(2) Prioritize (LO, MED, HI) the test items on each list, based on schedule, resources, and risk of not testing each item.
(3) For each list, create a coverage matrix which will later (after the actual test cases are identified) show the mapping between test items and test cases (i.e., which test cases cover which test items).
(4) For critical software, create a requirements tracing matrix.

When assessing risk, be aware that an untested condition could prove inconsequential for one customer and disastrous for another because of their different usage patterns.

For critical software, the IEEE/ANSI Standard for Software Verification and Validation Plans (Std 1012-1986) requires:

- requirements traceability analysis
- design traceability analysis
- code traceability analysis.

Requirement	Functional design	Internal design	Code	Tests
Restaurant has two ordering stations	Mgmt screen #2	Page 45	Line 12485	34, 57, 63
A waiter may order from any station	Order screen	Page 19	Line 6215	12, 14, 34, 57, 92
Any customer at a table may request a separate check	Order screen	Page 39	Line 2391	113, 85
A customer may get checks from more than one station	Check printing	Page 138	Lines 49234, 61423	74, 104

Figure 10.1 Example of a requirement tracing matrix. (© 1993, 1994 Software Development Technologies)

These analyses are embodied in the requirements tracing matrix, the purpose of which is to help ensure that for critical software, nothing falls through the cracks (see Figure 10.1). Ultimately there is a way of mapping for each requirement which tests cases exist to cover that requirement and the functional design.

The requirements tracing matrix links each requirement to its destination in the functional design, to its support in the internal design, to its support in the code, and to the set of tests for that requirement. References are re-verified when a change occurs.

For a typical real-world product to be tested, there are usually many test design specifications. The test design specification is a kind of umbrella document that helps identify test cases. The same test case may be identified in more than one test design specification. There are one or more test case specifications per test design specification. In practice, many organizations combine these two documents, eliminating the need for the more detailed test case specifications.

Deliverable: test design specification

(IEEE/ANSI, 1983 [Std 829-1983])

Purpose:

- To specify refinements of the test approach and to identify the features to be covered by the design and its associated tests. It also identifies the test cases and test procedures, if any, required to accomplish the testing and specifies the feature pass/fail criteria.

Outline:

- Test design specification identifier
- Features to be tested
- Approach refinements
- Test case identification
- Feature pass/fail criteria

Deliverable: test case specification

(IEEE/ANSI, 1983 [Std 829-1983])

Purpose:

- To define a test case identified by a test design specification. The test case spec documents the actual values used for input along with the anticipated outputs. It identifies any constraints on the test procedures resulting from use of that specific test case. Test cases are separated from test designs to allow for use in more than one design and to allow for reuse in other situations.

Outline:

- Test case specification identifier
- Test items
- Input specifications
- Output specifications
- Environmental needs
- Special procedural requirements
- Intercase dependencies

Implementation is the process of translating each test case specification into ready-to-execute test cases. The deliverables of implementation are:

- test cases, test data
- test procedure specifications
- completed function-coverage matrix

- completed requirements-coverage matrix
- for critical software, a completed requirements tracing matrix.

The test procedure specification explains step by step how to set up, how to start the tests, how to monitor the test run, and how to restart the tests if they have been suspended. A good written specification is important, so information on running this particular test library isn't just in someone's head.

Deliverable: test procedure specification

(IEEE/ANSI, 1983 [Std 829-1983])

Purpose:

- To identify all steps required to operate the system and exercise the specified test cases in order to implement the associated test design. The procedures are separated from test design specifications as they are intended to be followed step by step and should not have extraneous detail.

Outline:

- Test procedure specification identifier
- Purpose
- Special requirements
- Procedural steps

Test execution

A test execution overview is as follows. Tasks include:

- test case selection
- pre-run setup, execution, post-run analysis
- recording activities, results, and incidents
- determining whether failures are caused by errors in the product or in the tests themselves
- measuring internal logic coverage.

Test execution is the process of executing all or selected test cases and observing the results. Some organizations (those having a more mature test process) require that certain criteria be met before a product is eligible for entry into test (execution). The rationale for such criteria is to prevent the premature testing of a product that isn't ready for testing, and the wasting of time in both development and testing groups.

Example: requirements for entry to test (execution)

- The product is fundamentally complete.
- The product being submitted for test is a candidate for release to customers.
- All appropriate verification activities for this version of the product have been completed.
- All specifications in the derivation chain of the code are approved and frozen with respect to this version of the product.
- A set of "acceptance tests" selected by the test group have been executed without incident. Such acceptance tests are a proper subset of the test repository, and their incident free execution should indicate that the product is sufficiently reliable to make a complete test possible.

The key test execution deliverables are test logs, test incident reports, and the logic coverage report (tool output).

Usability testing may be performed numerous times throughout the development and test cycle. Usability testing is considered a validation activity because it employs real users in a product execution environment (real, simulated or mockup). As such, there may be many usability testing sessions, each of which should produce one test log report and any number of test incident reports.

The test log is the document in which we save the details about the execution of the tests. What do we save? What do we document? How long do we keep it around? It is the kind of information that is useful to refer to if we need to demonstrate whether a problem is a day-one problem or a regression.

Deliverable: test log

(IEEE/ANSI, 1983 [Std 829-1983])

Purpose:

- To provide a chronological record of relevant details about the execution of tests.

Outline:

- Test log identifier
- Description
- Activity and event entries

An incident report is another name for a defect (i.e., problem, bug) report. The most important part of it is the incident description, which should not just describe what happened but should always compare expected results with actual results.

Deliverable: test incident report

(IEEE/ANSI, 1983 [Std 829-1983])

Purpose:

- To document any test execution event which requires further investigation.

Outline:

- Test incident report identifier
- Summary
- Incident description
- Impact

Any incident report, regardless of origin (tester, internal user, customer, etc.) should be assigned a degree of severity by the originator. Severity measures the actual or anticipated impact on the user's operational environment. Standard definitions for severity should be established. No less than three and no more than five levels of severity are recommended.

The notion of severity is valuable in establishing service priorities, in gauging a product's release readiness, and in defining numerous quality metrics. From the point of view of development, the severity of a confirmed problem is a function of two things: the probability that the problem will occur, and the impact of the problem when it does occur. From the point of view of the user who has experienced the problem, usually only the latter is relevant.

Test evaluation

A test evaluation overview is as follows:

- test coverage evaluation
- product error evaluation
- test effectiveness evaluation.

Test coverage evaluation is the process of assessing the thoroughness of the collective set of test cases for the product and deciding whether or not to develop additional tests. The thoroughness assessment is based on current test coverage at various levels; function coverage (use function coverage matrix), requirements coverage (use requirements coverage matrix), logic coverage (using coverage measurement results). The specific deliverables of this effort are additional tests (usually internals-based) as appropriate.

Function coverage and requirements coverage are known prior to test execution. Test coverage analysis cannot be fully completed until test execution provides information on logic coverage. If more tests are required, we loop back into the "detailed design" stage of test development.

Product error evaluation is the process of assessing the quality of the product, with respect to the test execution, and deciding whether or not to develop additional tests. Usually 20% of the code has the lion's share of the errors. The quicker this 20% is located the better, because it is a good indicator of errors remaining. This assessment is based on the number of detected errors, their nature and severity, the areas of the product in which errors were detected and the rate of error detection. The specific deliverables of this effort are additional tests (that may have any test basis), if necessary.

Test effectiveness evaluation is the process of assessing the overall effectiveness of the current testing effort relative to the test completion criteria and deciding whether to stop testing or to add more tests and continue. The effectiveness assessment is based jointly on test coverage evaluation and product error evaluation.

At this point the key issues are:

- Do we decide to stop or continue testing?
- What additional tests are needed, if the decision is to continue?
- How do we create the test summary report, if the decision is to stop?

When do we stop?

The classic test completion criteria are that the clock runs out (i.e., the allocated testing time elapses), or all tests run to completion without detecting any errors.

Fortunately, there are meaningful and useful completion criteria. Ideally they should be based on all of the following components:

(1) The successful use of specific test case design methodologies.
(2) A percentage of coverage for each coverage category.
(3) The rate of error detection (i.e., the number of errors detected per unit of testing time) falls below a specified threshold. A criteria based on the number of detected errors should be qualified by the error severity levels.
(4) The detection of a specific number of errors (a percentage of total errors estimated to exist), or a specific elapsed time.

A model for tracking test execution status

Count the number of test cases in each of the following categories:

- ***Planned*: Test cases that planned to be developed.**
- ***Available*: Planned test cases that are available for execution. (Available is less than or equal to planned.)**
- ***Executed*: Available test cases that have been executed. (Executed is less than or equal to available.)**
- ***Passed*: Executed test cases whose most recent executions have detected no errors. (Passed is less than or equal to executed.)**

Plot these four numbers periodically (weekly, daily, etc.) against time and analyze the trends.

If test cases are divided into these four categories, a matrix can be set up on a product-by-product basis, and tracked over time, or tracked by ratios like passed/planned ratios. We could, for instance, decide to ship the product only when the passed to plan ratio is over 97%. Graphics on test execution status can provide all sorts of useful curves that can be analyzed to provide good practical test management information (see Figure 10.2).

The final document is the test summary report, in which the results of validation activities are summarized; what was planned, what was achieved in terms of coverage, how many defects were found, and of what severity. It can also be regarded as a report for management on testing activity.

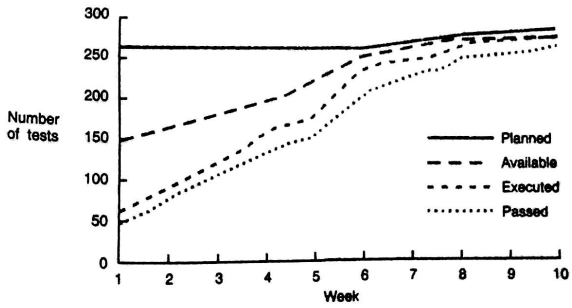


Figure 10.2 Sample test execution graph. (© 1993, 1994 Software Development Technologies)

Deliverable: test summary report

(IEEE/ANSI, 1983 [Std 829-1983])

Purpose:

- To summarize the results of the testing activities associated with one or more test design specifications and to provide evaluations based on these results.

Outline:

- Test summary report identifier
- Summary
- Variances
- Comprehensive assessment
- Summary of results
- Evaluation
- Summary of activities
- Approvals

A testing orphan – user manuals

Every software product that is delivered to a customer consists of both the executable code and the user manuals. A product's documentation is no less important than its code, because its quality is a significant factor in the success or failure of a product. From the point of view of the user, if the manual says do something, and the user follows these instructions and it doesn't work, it doesn't really help that the code is in fact right. Software and testing literature are predominantly silent on the testing of manuals, probably because manuals are usually produced by technical writers (not developers or testers) using life cycles, deliverables, and terminology for which there are no standards.

"User Documentation Evaluation" is defined as an optional task by IEEE/ANSI Standard for Software Verification and Validation Plans (Std 1012-1986). The task can occur in all phases of the development cycle. Documentation may include user manuals or guides, as appropriate to the project.

Examine draft documents during the development process to ensure correctness, understandability, and completeness. Treat manuals as important elements of a product. They should be subjected to a comprehensive testing process, using the concepts and methods of verification, including plans and reports.

Consider employing a critical review process. If the manuals are perceived as equal in importance to the code, then formal inspections are appropriate. Inspect manuals for correctness, understandability, and completeness.

To the degree that the manuals are perceived as less important than the code, then use less formal methods. Use the generic document verification checklist in Appendix B when reviewing manuals.

If the functional design specification is faithfully maintained, then it is the test basis for the manuals because the primary test objective for manuals is to find discrepancies between them. All examples in the manuals should be performed as part of the manuals testing effort to determine if they work as described.

Product release criteria

While most organizations have some kind of release criteria, all too often they are very informal. As companies mature, release criteria become more specific and more measurable. Releasing a new version of a product to customers (any version, not just the first) involves more than satisfactory completion of validation testing. Some typical release criteria are:

- all components of the end product, including user documentation, are complete and tested;
- software release and support policy is defined and understood;
- software manufacturing/distribution is ready;
- software support is ready;
- customers know how to report problems.

Summary of IEEE/ANSI test related documents

The following is a summary of all of the documents used in test planning and specification and how they relate to each other and the various testing activities and the respective standards. See Figures 10.3 and 10.4.

Documentation structure for test planning and specification

SQAP: Software Quality Assurance Plan (IEEE/ANSI, 1989 [Std 730-1989]). One per software product to be tested.

SVVP: Software Verification and Validation Plan (IEEE/ANSI, 1986 [Std 1012-1986]). One per SQAP.

VTP: Verification Test Plan. One per verification activity.

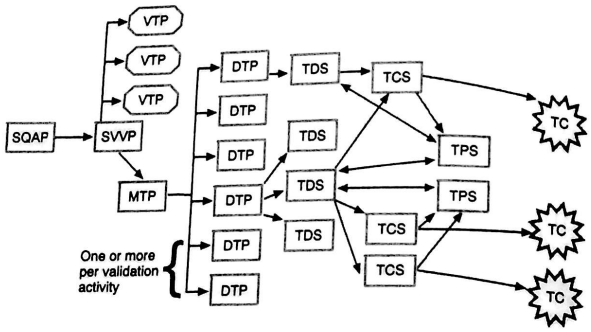


Figure 10.3 Documentation structure for test planning and specification. (© 1993, 1994 Software Development Technologies)

MTP: (Master Validation) Test Plan (IEEE/ANSI, 1983 [Std 829-1983]). One per SVVP.
DTP: (Detailed Validation) Test Plan (IEEE/ANSI, 1983 [Std 829-1983]). One or more per activity.
TDS: Test Design Specification (IEEE/ANSI, 1983 [Std 829-1983]). One or more per DTP.
TCS: Test Case Specification (IEEE/ANSI, 1983 [Std 829-1983]). One or more per TDS/TPS.
TPS: Test Procedure Specification (IEEE/ANSI, 1983 [Std 829-1983]). One or more per TDS.
TC: Test Case. One per TCS.

Documentation structure for test reporting

VTR: Verification Test Report. One per verification activity.
TPS: Test Procedure Specification. (IEEE/ANSI, 1983 [Std 829-1983]).
TL: Test Log (IEEE/ANSI, 1983 [Std 829-1983]). One per testing session.
TIR: Test Incident Report (IEEE/ANSI, 1983 [Std 829-1983]). One per incident.
TSR: Test Summary Report (IEEE/ANSI, 1983 [Std 829-1983]). One.

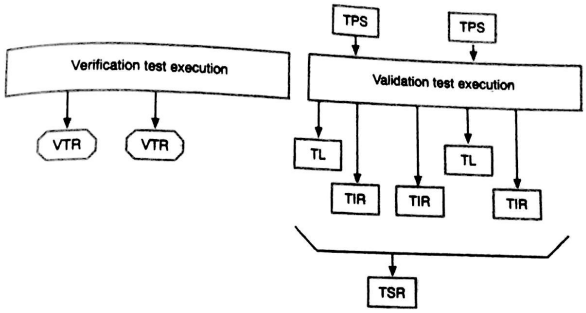


Figure 10.4 Documentation structure for test reporting. (© 1993, 1994 Software Development Technologies)

Life-cycle mapping of tasks and deliverables

This section maps each input, task, and deliverable of testing into its appropriate phase of a software life cycle with the following phases:

- concept
- requirements
- functional design
- internal design
- coding
- integration and test
- operation/maintenance.

The various phases of a life-cycle model may overlap to a considerable extent, but phase completions (approval of deliverables, signoffs, etc.) should occur in a specified sequence. Many of the testing tasks may actually span multiple phases, but they are shown in the earliest possible start phase. Testing deliverables are shown in the latest possible completion (due) phase.

The life-cycle scenario presented here presumes the full form of testing (where testing involvement begins no later than the requirements phase).

Concept phase

(1) *Inputs to testing*
  (i) Informal project discussions
(2) *Testing tasks*
  (i) Strategic planning
  (ii) Learn as much as possible about the product and project
(3) *Testware deliverables*
  none

Requirements phase

(1) *Inputs to testing*
  (i) Software quality assurance plan (optional, from SQA)
  (ii) Requirements (from development)
(2) *Testing tasks*
  (i) Plan (for verification and validation separately)
  (ii) Analyze the requirements
  (iii) Verify the requirements
  (iv) Begin to identify, itemize, and design requirements-based tests and develop a requirements coverage or tracing matrix
(3) *Testware deliverables*
  (i) Software V&V plan
  (ii) Verification test plan (for requirements)
  (iii) Verification test report (for requirements)

Functional design phase

(1) *Inputs to testing*
  (i) Functional design specification (from development)
(2) *Testing tasks*
  (i) Plan for functional design verification, validation
  (ii) Analyze the functional design specification
  (iii) Verify the functional design specification
  (iv) Begin performing usability tests
  (v) Begin to identify, itemize, and design function-based tests and to develop a function coverage matrix
  (vi) Begin implementation of requirements-based and function-based tests

(3) *Testware deliverables*
  (i) (Master validation) test plan (IEEE/ANSI, 1983 [Std 829-1983])
  (ii) Verification test plan (for functional design)
  (iii) Verification test report (for functional design)

Internal design phase

(1) *Inputs to testing*
  (i) Internal design specification (from development)
(2) *Testing tasks*
  (i) Plan for internal design verification
  (ii) Analyze the internal design specification
  (iii) Verify the internal design specification
  (iv) Begin to identify, itemize, and design internals-based tests
(3) *Testware deliverables*
  (i) (Detailed Validation) Test Plans (IEEE/ANSI, 1983 [Std 829-1983]) – one or more validation activity
  (ii) Verification test plan (for internal design)
  (iii) Verification test report (for internal design)
  (iv) Test design specification (IEEE/ANSI, 1983 [Std 829-1983])

Coding phase

(1) *Inputs to testing*
  (i) Code (from development)
(2) *Testing tasks*
  (i) Plan for code verification
  (ii) Analyze the code
  (iii) Verify the code
  (iv) Design externals-based tests
  (v) Design internals-based tests
(3) *Testware deliverables*
  (i) Test case specifications (IEEE/ANSI, 1983 [Std 829-1983])
  (ii) Requirements coverage or tracing matrix
  (iii) Function coverage matrix
  (iv) Test procedure specifications (IEEE/ANSI, 1983 [Std 829-1983])
  (v) Verification test plan (for code)
  (vi) Verification test report (for code)
  (vii) Validation testware (function and system tests)

Integration and test phase

(1) *Inputs to testing*
  (i) Drafts of user manuals
  (ii) Software to be tested
  (iii) Final user manuals
  (iv) Test item transmittals (IEEE/ANSI, 1983 [Std 829-1983])

(2) *Testing tasks*
  (i) Planning
  (ii) Review module and integration testing performed by development
  (iii) Perform function tests
  (iv) Perform system tests
  (v) Review draft and final versions of user manuals

(3) *Testware deliverables*
  (i) Test logs (IEEE/ANSI, 1983 [Std 829-1983])
  (ii) Test incident reports (IEEE/ANSI, 1983 [Std 829-1983])
  (iii) Test summary report (IEEE/ANSI, 1983 [Std 829-1983])

Operation/maintenance phase

(1) *Inputs to testing*
  (i) (see note)
  (ii) Confirmed problem reports (from any source)

(2) *Testing tasks*
  (i) Monitor acceptance testing
  (ii) Develop new validation tests for confirmed problems
  (iii) Evaluate continued effectiveness of all tests

(3) *Testware deliverables*
  (i) Updated test repository

Note: The software life cycle is an iterative process. After the initial release of a product, any change to the product should require that development and testing activities revert to the life-cycle phase that corresponds to the type of change made. For example, if a new function is added to the product (a new function not instigated by a requirements change), then a new functional design specification is required, and the process should revert to the functional design phase and continue sequentially thereafter. In other words, all development and testing activities do not get lumped into the operation/maintenance phase just because the product has been released to customers.

References

IEEE/ANSI (1983). IEEE Standard for Software Test Documentation, (Reaff. 1991), IEEE Std 829-1983.

IEEE/ANSI (1986). IEEE Standard for Software Verification and Validation Plans, (Reaff. 1992), IEEE Std 1012-1986.

IEEE/ANSI (1988). IEEE Standard for Software Review and Audits, IEEE Std 1028-1988.

IEEE/ANSI (1989). IEEE Standard for Software Quality Assurance Plans, IEEE Std 730-1989.

Chapter 11
Software testing tools

The use of testing tools can make testing easier, more effective and more productive. It is no coincidence that one of the first stated goals of attendees at testing courses is: What tools should we buy for my organization?

A wide variety of computer-aided software testing (CAST) tools are available, addressing many aspects of the testing process. Their scope and quality vary widely, and they provide varying degrees of assistance.

If we are to benefit from one of the major sources of leverage in the testing effort, a strategy for evaluation, acquisition, training, implementation, and maintenance is essential. It is an area where independent expertise can be enormously beneficial.

Categorizing test tools

There are a number of ways to categorize testing tools:

(1) *by the testing activity or task in which it is employed* – activities in this case include code verification, test planning, test execution;
(2) *by descriptive functional keyword* – the specific function performed by the tool such as capture/playback, logic coverage, comparator;
(3) *by major areas of classification* – a small number of high-level classes or groupings of tools.

Each class contains tools that are similar in function or other characteristics. Examples of these are test management tools, static analysis tools, and simulators.

The present chapter will focus on the activities with which the tool is associated, namely:

- reviews and inspections
- test planning

- test design and development
- test execution and evaluation
- test support.

We have chosen this approach of categorization for two reasons. First, it is closest to the point of view of the tester – what the tester is doing and when. Second, it is consistent with and based on the testing standards.

A listing of specific tools by descriptive function is included in Appendix G. In general, at the front end of the testing process there are fewer specialized tools available than at the back end.

Tools for reviews and inspections

Tools for reviews and inspections are the tools that assist with performing reviews, walkthroughs, and inspections of requirements, functional design, internal design, and code. Some tools are designed to work with specifications but there are far more tools available that work exclusively with code.

The types of tools required are:

- complexity analysis
- code comprehension
- syntax and semantic analysis.

Complexity analysis

Experienced programmers know that 20% of the code will cause 80% of the problems, and complexity analysis helps to find that all-important 20%. The McCabe Complexity Metrics were originally published in 1982 in the NBS (National Bureau of Standards) publication, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric."

Complexity metrics identify high risk, complex areas. The cyclomatic complexity metric is based on the number of decisions in a program. It is important to testers because it provides an indication of the amount of testing (including inspections) necessary to practically avoid defects. In other words, areas of code identified as more complex are candidates for inspections and additional tests. There are other types of complexity metrics (e.g., from McCabe and Halstead), which in general are an indicator of program testing cost/schedule and number of defects in the program.

Code comprehension

Code comprehension tools help us understand unfamiliar code. They help us to understand dependencies, trace program logic, view graphical representations of the program, and identify dead code. They can be successfully used to identify areas that should receive special attention, such as areas to inspect.

There is a considerable amount of time spent in preparing for a code

inspections meeting. This requires extensive analysis, comprehension, and reverse engineering, all of which are made easier by code comprehension tools.

Syntax and semantic analysis
Syntax and semantic analysis tools perform extensive error checking to find errors that a compiler would miss, and are sometimes used to flag potential defects before or sometimes during formal testing.

These tools are language (and sometimes dialect) dependent. With the programming language C, for example, since there are a variety of dialects, these tools can often be configured by dialect. They parse code, maintain a list of errors, and provide build information. The parser can find semantic errors as well as make an inference as to what is syntactically incorrect.

Tools for test planning

The purpose of test planning is to define the scope, approach, resources (including tools), and schedule of testing activities. The test plan provides the foundation for the entire testing process, and, if this sounds like a cerebral activity, it should. Tools don't eliminate the need to think. As useful as capture/playback tools are (see the section on Test execution and evaluation below), they do not replace the need for sound test planning and design.

Perhaps the biggest help here comes from standards. The IEEE/ANSI Standard for Software Test Documentation (Std 829-1983) describes the purpose, outline, and content of the test plan, and the appendix of the standard includes examples taken from commercial data processing. Although a few tools have incorporated templates for test plans into them, many companies have found it useful to simply have someone enter the outline for test plans found in the IEEE/ANSI standard into an accessible edit file.

There are useful commercial tools that determine actual project staffing and schedule needs for adequate product testing. Often people rightly complain that schedules are predetermined by upper management, and the use of such a tool can provide an objective view of the realities of the project.

The types of tools required for test planning are:

- templates for test plan documentation
- test schedule and staffing estimates
- complexity analyzer.

Tools that help with reviews and inspections will also help with test planning, i.e., tools that identify complex product areas can also be used to locate areas that should impact planning for additional tests based on basic risk management.

Tools for test design and development

Test design is the process of detailing the overall test approach specified in the test plan for software features or combinations of features, and identifying and prioritizing the associated test cases. Test development is the process of translating the test design into specific test cases.

Like test planning, there's not a lot of help from test tools for the important, mostly mental process of test design. However, tools from the test execution and evaluation category, for example, capture/playback tools, assist with the development of tests, and are most useful as a means of implementing test cases that have been properly planned and designed.

The types of tools required for test design and development are:

- test data generator
- requirements-based test design tool
- capture/playback
- coverage analysis.

Once again, standards help. The IEEE/ANSI Standard for Software Test Documentation (Std 829-1983) describes the purpose, outline, and content of the test design specification, and the appendix of the standard also includes examples taken from commercial data processing.

Although a few tools have incorporated templates for the test design specification into them, many companies have found it useful to simply have someone enter the outline for test design specifications found in the IEEE/ANSI standard into an accessible edit file.

A useful type of tool in this category is a test data generation tool, which automates the generation of test data based on a user-defined format, for example, automatically generating all permutations of a specific, user-specified input transaction.

A tool that has not yet achieved widespread practical use is a requirements-based test design tool. Based on the assumption that faulty requirements can account for over 80% of the cost of errors, this highly disciplined approach based on cause-effect graph theory is used to design test cases to ensure that the implemented system meets the formally specified requirements document. This approach is for those who desire a disciplined, methodical, rigorous approach.

Test execution and evaluation tools

Test execution and evaluation is the process of executing test cases and evaluating the results. This includes selecting test cases for execution, setting up the environment, running the selected tests, recording the execution activities,

analyzing potential product failures, and measuring the effectiveness of the effort. Tools in the evaluation category assist with the process of executing test cases and evaluating the results.

The types of tools required for test execution and evaluation are:

- capture/playback
- coverage analysis
- memory testing
- simulators and performance.

Capture/playback

There is perhaps no chore more boring to the experienced tester than having to repeatedly re-run manual tests. Testers turn to capture/playback tools to automate the execution of tests, in other words, to run tests unattended for hours, overnight, or 24 hours a day if desired.

Capture/playback tools capture user operations including keystrokes, mouse activity, and display output. These captured tests, including the output that has been validated by the tester, form a baseline for future testing of product changes. The tool can then automatically play back the previously captured tests whenever needed and validate the results by comparing them to the previously saved baseline. This frees the tester from having to manually re-run tests over and over again when defect fixes and enhancements change the product.

Capture/playback tools can be classified as either native or non-intrusive. The native (sometimes called intrusive) form of capture/playback is performed within a single system. Both the capture/playback tool and the software being tested reside in the same system, i.e., the test tool is "native" to the system under test. It is sometimes called intrusive because the capture/playback software is distorting the operating performance to some extent, though for most software testing, this distortion is irrelevant.

The non-intrusive form of capture/playback requires an additional hardware system for the test tool. Usually the host system (containing the software under test) has a special hardware connection to the capture/playback tool, and this enables the capture/playback system to perform its required functions transparently to the host software. The best non-intrusive tools are platform and operating system independent.

There are three forms of capture/playback, listed in order of least to most expensive:

(1) *native/software intrusive* (introduces distortion at software level within the system under test);
(2) *native/hardware intrusive* (introduces distortion at hardware level only);
(3) *non-intrusive* (no distortion).

The most common type in use is native/software intrusive. Non-intrusive is typically used when the product being tested is itself an integrated hardware and software system where the introduction of additional internal hardware or software cannot be tolerated, e.g., real-time embedded systems. Since most software testing does not have this constraint, native/software intrusive is usually the cost-effective solution used by most organizations.

Choosing the right capture/playback tool turns out to be one of the most important and also most complex decisions an organization must make regarding testing. Unfortunately, capture/playback tool buyers are often forced to worry about such things as GUI test synchronization, proprietary testing languages, variable execution speed control, portability, multitasking testing, client/server, non-compare filtering, and non-intrusive testing. The best tools combine functionality with ease of use and mask many of the complexities of the tool's internal operation.

Coverage analysis

Coverage analyzers provide a quantitative measure of the quality of tests. In other words, they are a way to find out if the software is being thoroughly tested. This tool, essential to all software test organizations, tells us which parts of the product under test have in fact been executed (covered) by our current tests. They will tell us specifically what parts of the software product are not being covered, and therefore require more tests.

Some companies argue that it is not necessary to achieve full statement coverage, that is, to execute all of the statements within the product prior to release. They seem to think it is all right to expect customers to be the first to execute their code for them. Maybe these companies belong in another line of work. If we aren't measuring coverage, we do not have a handle on the job we are doing as testers.

Almost all structural tools run the source code into a preprocessor so that it can keep track of the coverage information. The problem is we now have a new source that's bigger than the old one so our object module is going to grow in size. The other possible problem is that performance may be impacted because we now have a different program than we did before. However, the final version of the software as delivered does not include the above preprocessing step, and therefore does not suffer this size and performance penalty.

There are many varieties of coverage, including statement, decision, condition, decision/condition, multiple condition, and path. As a minimum, the place to start is to make sure each statement in the program has been tested, and that each decision has taken on all possible outcomes at least once.

Memory testing

Whether being called bounds-checkers, memory testers, run-time error detectors, or leak detectors, in general the tools in this category include the ability to detect:

- memory problems
- overwriting and/or overreading array bounds
- memory allocated but not freed
- reading and using uninitialized memory.

Errors can be identified before they become evident in production and can cause serious problems. Detailed diagnostic messages are provided to allow errors to be tracked and eliminated.

Although memory testing tools tend to be language and platform specific, there are several vendors producing tools for the most popular environments. The top tools in this category are non-intrusive, easy to use, and reasonably priced. We put them in the "Just Do It" category, especially considering the alternative, i.e., shipping low-quality applications.

Test case management

The need for a test case management tool can creep up on us. We begin using a capture/playback tool to build and automate our tests. Then one day we wake up and find we have thousands of disorganized tests that need to be managed.

The best test case managers:

- provide a user interface for managing tests;
- organize tests for ease of use and maintenance;
- start and manage test execution sessions that run user-selected tests;
- provide seamless integration with capture/playback and coverage analysis tools;
- provide automated test reporting and documentation.

Why is test case management in a separate category, i.e., why isn't this incorporated into existing capture/playback tools? The bad news is – we're not there yet. The good news is that several of the leading tool vendors claim that they are working hard to accomplish this, and within the next year or so we should see several viable offerings.

Simulators and performance

Simulators take the place of software or hardware that interacts with the software to be tested. Sometimes they are the only practical method available for certain tests; for instance, when software interfaces with uncontrollable or unavailable hardware devices. They are frequently used to test telecommunications application programs, communications access methods, control programs, and networks.

Simulators also allow us to examine system performance. In general, performance tools help to determine what the software and system performance capabilities are. In practice, it is sometimes hard to find the line that distinguishes a simulator from a performance tool.

Finally, there are tools available for automated multi-user client/server load testing and performance measurement. These tools make it possible to create, control, and analyze the performance testing of client/server applications – before these applications go on line.

Software testing support tools

The test support category includes tools that, while not at the heart of the test process, lend overall support to the overall test effort. When these tools are of poor quality or do not exist, the professional tester suffers.

The types of tools required for test support are:

- problem management
- configuration management.

Problem management

Problem management tools are sometimes called defect tracking tools, bug management tools, incident control systems, etc., and are used to record, track, and generally assist with the management of defects and enhancements throughout the life cycle of software products.

Although many companies spend large sums developing home-grown problem management systems, there are tool vendors who now specialize in creating such systems across a variety of platforms. The best problem management tools are easy to customize for particular environments, and offer as standard features the capability to easily:

- and quickly submit and update defect reports;
- generate pre-defined or user-defined management reports;
- selectively notify users automatically of changes in defect status;
- provide secured access to all data via user-defined queries.

Configuration management

Configuration management (CM) is the key to managing, controlling, and coordinating changes to documents, and whatever else we really care about. CM tools assist the version control and build management process (see Chapter 6).

Besides problem management and configuration management, there are many tools not related to testing that in one way or another support the test process. These include project management tools, data base management software, spreadsheet software, and word processors.

Tool acquisition

The issues to be addressed in the acquisition of tools are largely good management common sense, but they are notoriously hard to implement. Far too often tools are purchased on an *ad hoc*, solve-the-short-term-crisis basis, and as a consequence end up on the shelf gathering dust and regarded as one more expensive experiment.

Making decisions about the acquisition of tools should involve some form of cost/benefit analysis, however simple. Tool vendors are naturally eager to tell us what their tool will do, and how it will solve our particular problems. The question we have to ask is: "At what cost?"

The important thing about cost is to establish true cost – meaning total cost or even lifetime cost. This will be a guesstimate, but it's a lot better than nothing, and as usual the purchase or license price is only the beginning. Additional costs are incurred in selection, installation, operation, training, maintenance and support, and the general cost of reorganizing procedures.

Tools that support testing processes which are already in place can be implemented with much less pain, human and financial, than those which will require an organizational clean sweep.

The scope of initial use needs to be established. Should the new tool be started with a single group, or even selected individuals, or should it be implemented organization-wide immediately?

Management support at the early stages is critical to ensure that the people who are actually working with the new tool can focus their attention on getting up and running and being productive as early as possible.

The careful choice and implementation of tools that support each other and have a synergistic effect is another major source of productivity.

The difficulty is finding the right tools from the right vendors. It is a major undertaking to produce and maintain a current evaluation of available testing tools and their capabilities. There are several hundred testing tools on the market, from companies that vary widely in size, installed customer base, product maturity, management depth, and understanding of testing and tools.

For more detail on tools and the tool vendor selection process, see Appendix G.

Independent expert advice at the early stages, preferably from the stage of evaluating processes and deciding what kinds of tools would be useful right up to the implementation stage, is invaluable. There is no doubt that testing can be easier, more effective, and more productive by using tools. By taking the proper steps, we can prevent an expensive new tool from becoming shelfware.

Questions before tool acquisition

There are a few questions that we recommend you answer as part of implementing an effective tools program:

- How do the tools fit into and support our test process?
- Do we know how to plan and design tests? (Tools do not eliminate your need to think, to plan, to design.)
- Who will be responsible for making sure we get the proper training on our new tool?
- Who will promote and support tool use within the organization on an ongoing basis?

Reference

IEEE/ANSI (1983). IEEE Standard for Software Test Documentation, (Reaff. 1991), IEEE Std 829-1983.

Chapter 12
Measurement

There are a number of big questions for testing – questions about product quality, risk management, release criteria, the effectiveness of the testing process, and when to stop testing.

Measurement provides answers. But once we start to think about what can be measured, it's easy to be overwhelmed with the fact that we could measure almost anything. However, this isn't practical (for the same reason that we can't test everything), and we have to create priorities for measurement based on what measures are critical and will actually be used once we have them.

> "Not everything that counts can be counted, and not everything that can be counted counts."
>
> Albert Einstein

Measurement for the sake of measurement can result in wasted effort. We should ask: "Is it useful? How will we profit from this measure? Is there an important question we could answer satisfactorily if we had reliable measurements in a particular area?" Or perhaps there is an important question we can't ask until we have a particular measurement.

Measurement provides answers

If our planning and subsequent activities are to be effective, they must be developed on a reliable, factual basis. How long should the testing take? How efficient is our testing? How good is our test library? Is it worth doing verification as opposed to validation? How thorough is our validation testing?

Based on past experience, what sort of state is the product likely to be in when it is declared "ready for testing?" What kinds of errors are we likely to find, how many, and where? How many errors probably remain in the product after testing? How is the product doing in test and production compared to other products on the market?

Measuring the number and types of errors detected during verification provides a measure of the efficiency of verification. Verification is expensive, and while we may be convinced that it is cost effective, this effectiveness will often need to be justified to others. How many errors are we picking up in validation that could have been found in verification?

Measuring validation test coverage (requirements, function, and logic coverage) provides quantitative assessments of the thoroughness and comprehensiveness of the validation tests. How much of the product are we testing? How good is our test library?

Measuring/tracking test execution status shows the convergence of key test categories (planned, available, executed, passed) and provides quantitative information on when to stop testing. How many of our tests are planned? How many are available? How many have been executed? How many have passed? When can we (reasonably) stop?

Program complexity provides answers useful when planning the size of the test effort and estimating the number of errors before testing begins.

Measuring and tracking of incident reports (by severity category) is a leading indicator of product quality. It provides an objective criteria for release readiness, a predictor of the number of remaining errors, and generally correlates to users' satisfaction with the product. When normalized, it provides a measure of product quality relative to other products.

When incident reports are not tracked, companies lose the handle on being able to fix problems responsibly. The backlog of incidents grows until it is so large that they no longer have a plan for managing it or getting it down to a reasonable size. Measurement provides an early warning that this situation is developing.

Customer-reported versus testing-reported incidents (only those diagnosed as confirmed errors) provide another measure of testing effectiveness. It is also important that testers have access to defects found by customers because these are a valuable basis for improving the test library (see Hetzel, 1993).

Useful measures

Measuring complexity

Generally, the more complex components of a program are likely to have more errors, and the more focused the testing effort must be to find them. There are plenty of good tools available off the shelf in this area, and it isn't necessary to understand exactly how these tools work to use them efficiently.

There are several well-known measures of program complexity. Most complexity measures can only be calculated after the program is written. Such measures are useful for certain validation tasks and defect prediction.

A simple measure of complexity that is fundamentally a measure of size is lines of code (LOC) or number of source statements, and can be counted in several different ways. A given line might be blank, a comment, one or more executable statements and/or data declarations. Also, there is the problem of comparing the LOC of programs written in different source languages. The simplest way to normalize the LOC is to generate the assembly-language equivalent program and then count LOC. We must decide how to count LOC and then standardize it within the organization. It is less critical to spend an inordinate amount of time debating exactly how to count than it is to count everything exactly the same way using the same tool.

An alternative to counting lines of code is function points. Like lines of code, function points are also a measure of size, effort, and complexity. Unlike lines of code, function points are derived from the user's perspective as detailed in functional specifications, and will stay constant independent of programming language. From the testing process perspective, function points have been used to estimate test cases required per function point and in measuring defects per function point.

In practice, debating lines of code versus function points often becomes a heated, religious discussion. Those that love function points see lines of code as the old, inaccurate approach with many significant shortcomings. The complaints about function points are that they are too abstract, do not relate as closely to what software developers actually produce, require training to learn how to count, and involve a process that seems complex to use to the uninitiated.

Both lines of code and function points today still have a place in software production and are worth considering. In the long term, as function points or a derivative become easy to understand, count, and use, they are likely to eventually completely replace the need to measure lines of code.

Another measure of complexity is McCabe's complexity metric which is the number of decisions (+1) in a program. An N-way branch is equivalent to N–1 decisions. Complexity across subprograms is additive. The result is a complexity number, which if it is above a certain limit, indicates the need for special attention, such as inspections or additional validation testing. Many of the leading tools provide an automatic capability for calculating the McCabe complexity metric.

Halstead's metrics are used for calculating program length (not to be confused with lines of code). Program length can be predicted before the program is written, and the predicted and actual values compare very closely over a wide range of programs. There are also formulas for predicting the number of bugs, programming effort, and time.

Measuring verification efficiency

The efficiency of verification activities is an important measure because verification is expensive. Verification test reports should contain lists of specific errors detected by each verification activity (see Chapter 10). Using subsequent error data from validation activities, one can count the errors detected and the errors missed (not detected) by verification.

Measuring test coverage

Coverage for requirements-based and function-based tests can be measured manually, using a requirements coverage/tracing matrix and a function coverage matrix. Logic coverage can be measured (practically) only with an automated tool. Measuring statement coverage is the most common practice in today's off-the-shelf coverage tools, and in general is a reasonable place to start.

Measuring/tracking test execution status

Test execution tracking is performed most simply by using a spreadsheet. The columns consist of a time stamp and the four test categories: planned, available, executed, passed. Each row is a periodic (e.g., weekly, daily) observation of the number of test cases in each category. The time between successive observations should be small enough to provide a sufficient number of observations to make any trends visible. The spreadsheet can be presented automatically in graphic form to make trends easier to interpret. A predetermined ratio of tests passed to tests planned (e.g., 98%) is often used as one criterion for release.

Measuring/tracking incident reports

Incident or "bug" reports can provide the basis of many valuable quality metrics for software. To realize the full potential of incident reports, the incident reporting process and its automated support system should follow a number of principles:

(1) There should be one and only one way to report an incident. Redundant reporting mechanisms create unnecessary work.
(2) There is a single, master repository for all incident reports. Fragmented, redundant, or incomplete data make it very difficult to obtain accurate, complete, and timely information.
(3) Every incident should be reported via the formal mechanism. Unreported incidents are never investigated. Informal reports often fall through the cracks, and their status cannot be tracked.
(4) Incident reports must rigorously identify the software configuration in which the incident occurred. Users must be able to dynamically obtain from the running system all necessary version information. Version identification can be trusted when it is obtained from the running system.

(5) Every user of a product, not just customers, should take the time to report incidents. This includes internal users, developers, testers, and administrators.

(6) After a new version of a product is baselined for formal testing and potential release (during the integration and test phase), all subsequent incidents should be formally reported. Problems detected before release are just as important as those detected after release.

(7) Every incident report should be investigated and then classified as one of the following:
   (i) user/operator error
   (ii) cannot reproduce the reported problem
   (iii) insufficient information
   (iv) documentation (user manual) error
   (v) request for change/enhancement
   (vi) confirmed product error
   (vii) other.

(8) Because a raw incident report describes the symptom of a problem (i.e., how the problem manifests itself to the user), the report must contain sufficient space for, or point to, the precise description of the real error. Many quality metrics are based on counts of confirmed errors, not incident reports. It must be possible to count the confirmed errors, not the symptoms of errors, in a specific product. This is not necessarily the product identified in the incident report. Multiple incident reports, each possibly describing different symptoms, should be able to point to the same underlying error without requiring a redundant description of the error in each incident report.

(9) For a confirmed problem, the report must also contain sufficient space for (or point to) the root cause categorization of the error. The best software engineering organizations have cultivated a culture within which a root cause analysis is performed for every confirmed error and then used in error prevention programs. Root causes are often rigorously categorized into a spectrum of standard causes.

(10) The incident reporting system should provide complete tracking information, from the time the report originates to the time it is formally closed. To manage the process, a state transition diagram should be developed to show all possible states of an incident report, the events which cause state changes, and the organizations authorized to make state changes.

(11) An incident report should not be considered closed until all work associated with the report is completed. In the case of a confirmed error, the correction should be available for customer use before the report is closed.

Test measures based on incident reports

The number of confirmed errors per 1,000 lines of code (errors/KLOC) is a common measure of error density that provides one indicator of product quality (or alternatively, the number of confirmed errors per function point). It is also an inter-product comparison of quality. The number of confirmed errors to date is itself a predictor of the number of remaining errors, because the number of undiscovered errors in a program is directly proportional to the number of errors already discovered.

A key component of release criteria usually states the number of confirmed, uncorrected errors (per severity level) that constitutes the limits of acceptability for release to customers. Comparing the number of confirmed errors reported by users and customers to the number reported by testing provides a measure of testing efficiency. The errors reported by users/customers also provide a basis for new test cases.

Other interesting measures

There are many measures of interest to software practitioners. A few of the common ones are:

(1) the age of a detected error;
(2) the response time to fix a reported problem;
(3) the percentage and frequency of detected errors by root-cause category;
(4) error removal efficiency;
(5) error cost:
   (i) cost of the failure to the user
   (ii) cost to investigate and diagnose
   (iii) cost to fix
   (iv) cost to retest
   (v) cost to release.

Recommendations

- Obtain consensus on the top three testing measurements to put in place in your organization.
- Put in place a capability for measuring/tracking test execution status based on the key test status model (planned, available, executed, passed).

- Locate a tool for measuring program complexity.
- Define the testing-related key release criteria and determine how to obtain the measurements.

References

Beizer, B. (1990). *Software Testing Techniques.* Van Nostrand Reinhold.
Capers Jones. (1991). *Applied Software Measurement.* McGraw-Hill.
Hetzel, W. (1993). *Making Software Measurement Work.* QED Publishing Group.
Humphrey, W.S. (1984). *Managing the Software Process.* Reading, MA: Addison-Wesley.
Kit, E. (1986a). *Testing C Compilers,* Computer Standards Conference.
Kit, E. (1986b). *State of the Art, C Compiler Testing.* Tandem Computers Technical Report.
IEEE/ANSI (1988a). IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.1-1988.
IEEE/ANSI (1988b). IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.2-1988.

PART IV
Managing test technology

"Losing the test technology game is losing the software quality game which means losing the software game. Losing the software game is losing the computer game which is losing the high-tech game which in turn means losing the whole post-industrial society ball of wax."

BORIS BEIZER, Losing it, An Essay on World Class Competition

Chapter 13
Organizational approaches to testing

Most of us live our lives in organizations that change routinely. We live in structures that are never completely finished. Software testing in particular is an area where companies try many different approaches to organization – partly because of the perplexity about which structures are most effective, i.e., lead to most effective relations between people.

Why do organizations exist? What are the most fundamental, basic building blocks of structural design? What are the advantages and disadvantages of the specific approaches used in the real world to organize software testing? How can we decide which approach to use at this stage of our company?

Organizations exist because there is a need for people as a group to behave predictably and reliably. To achieve results, people need to cooperate with each other. Organizations are created to support, and more to the point, coerce the activities people engage in. Good organizations minimize the inevitable conflict between the needs of the organization and the needs of the individual.

For example, some software organizations have successfully created a positive culture that encourages and rewards migration of individuals from product development groups into independent test development groups equally as much as migration from test to development. In fact, the best cultures reward those who over time complete the full loop – start by spending a few years in development, then spend a few years in test, then return to development.

We have known a few exceptional people to repeat this entire loop twice over a span of about 10 years. The amazing thing is that these people are viewed as the most valued, the most prized resource by managers in both product development and test development. Talk about job security! Why? What better person to have in development than someone who knows how to design a test library? This developer will know how to create software that passes tests, because he or she knows how to think like a tester. And what better person to have designing tests than someone who knows how developers think?

There are many managers who unfortunately only see testing as a training ground for development, not a place to put their best developers. Too bad. I count myself among the managers who *will not hire* a person to do testing unless that person is qualified today to join the development group responsible for developing the same product that he or she would be testing. The good testing manager will encourage such a person to get a few years' experience in product development, and only then to consider moving to the testing group. Radical thinking? We don't think so!

Organizing and reorganizing testing

Once we have personally experienced several reorganizations, we learn to appreciate the need for organizational change to be very carefully planned. Let's face it – organizational change creates stress, disruption, demoralization, and confusion to varying degrees. The following 2200-year old quotation helps us to see how long the problems surrounding organizational change have been recognized and articulated:

> "We trained hard ... but it seemed that every time we were beginning to form up into teams we would be reorganized ... I was to learn later in life that we meet any new situation by reorganizing, and a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency, and demoralization."
>
> Petronius Arbiter, 210 BC

Isn't it frightening how relevant this thought from Petronius remains today?

An organization is a system that combines materials, knowledge, and methods in order to transform various kinds of inputs into valued outputs. Organizational structure is the responsibilities, authorities, and relationships, arranged in a pattern, through which the organization performs its functions. Structural choices include reporting hierarchies, job descriptions, and goals.

A test organization (usually called a test group) is a resource or set of resources dedicated to performing testing activities. As shops grow, the need for a dedicated, independent testing function becomes a more apparent necessity. It takes unbiased people to produce an unbiased measurement – testing must be done independently if it is to be fully effective in measuring software quality.

Organizational structures are like software. Neither are ever really completely finished. With time, the demands (external and internal) on the structure change – the capability of the existing structure to meet these demands decreases – making organizational redesign a repetitive activity. Likewise, with time, the requirements for a software system change. As with organizations, no matter how well it is designed at the beginning, eventually, if it is to

remain useful, software must evolve. *We must leave behind the innocent delusion that once we understand the problem the software system is supposed to solve, we can go off alone to build and test it in peace.*

Test management is difficult. The manager of the test group must have the:

- ability to understand and evaluate software test process, standards, policies, tools, training, and measures;
- ability to maintain a test organization that is strong, independent, formal, and unbiased;
- ability to recruit and retain outstanding test professionals;
- ability to lead, communicate, support, and control;
- time to provide the care needed to manage test groups.

Senior managers must also have the ability to recruit and retain outstanding test managers. In fact, this is usually where the big mistakes are made. The senior managers do not understand the list above enough to know how to evaluate a potential test manager against these requirements. The result is that the wrong person is often promoted into test management!

When making organizational changes that affect testing, senior management needs to understand the impact the change will have on test management's ability to meet the above requirements. Plenty can go wrong when reorganizations occur without sufficient thought being given to the impact on testing. The dangers of having the wrong software testing structure include the following:

- Test independence, formality, and bias is weakened or eliminated.
- People in testing do not participate in reward programs.
- Testing becomes understaffed.
- Testing becomes improperly staffed with too many junior individuals.
- Testing is managed by far too junior managers.
- There is no leverage of test knowledge, training, tools, and process.
- Testing lacks the ability to stop the shipment of poor quality products.
- There is a lack of focused continuous improvement.
- Management lacks the bandwidth to manage testing groups.
- The quality focus is not emphasized.
- Test managers become demoralized owing to lack of career growth.

The above list can be used as a checklist of items to consider when planning organizational modifications. They can become questions; for example: Does making this change weaken the independence of testing? Does it hurt test's ability to get quality resources? etc.

Structural design elements

There is a surprisingly short list of basic building elements from which to construct a structure. The structural design elements are:

(1) *Tall or flat* – There may be many levels between the chief executive officer and the person on the shipping floor (this would be a tall organization), or there might be very few levels (a flat organization). In the last decade, flat has become more popular; managers finding themselves in the middle of a tall organization are particularly vulnerable to losing their jobs.
(2) *Market or product* – The organization may be structured to serve different markets or different products.
(3) *Centralized or decentralized* – The organization may be centralized or decentralized. This is a key question for the test organization. We will examine this in depth later in this chapter.
(4) *Hierarchical or diffused* – The organization may be hierarchical, that is, organized according to successively higher levels of authority and rank. Or it may be diffused, which is widely spread or scattered or matrixed.
(5) *Line or staff* – The organization will have a certain mix of line and/or staff roles.
(6) *Functional or project* – The organization may have functional or project orientations.

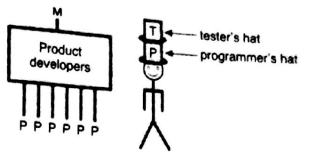
Combining these few design elements provides quite a variety of operating structural designs. Sometimes a variety of designs are implemented within the same company.

Approaches to organizing the test function

The above organizational basic design elements can be combined to produce many different test structures. There are seven approaches to organizing testing typically taken in practice that reflect the evolution of a maturing development organization. The following approaches to structure assume that unit testing should be done by product development. Therefore, the material that follows is about testing activities not related to unit testing, e.g., function testing, system testing, etc.

Approach 1. Testing is each person's responsibility

Approach 1 is what often occurs in the real world, especially when companies are small and little thought has been given to testing. As shown in Figure 13.1, there is a group of product developers whose primary responsibility is to



Advantages	Disadvantages
Seems on the surface like the most natural solution. (Evolves naturally out of the original product development team.)	Programmers are inherently incapable of effectively testing their own programs.

Figure 13.1 Approach 1: testing is each person's responsibility.

build a product. In this model, these product developers also are responsible for testing their own code. These people unfortunately must try their best to wear two hats, a programmer's hat and a tester's hat. They are responsible for function testing, system testing, and any other kind of testing that gets done.

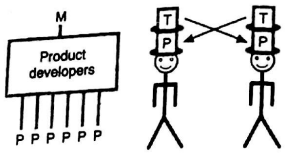
The problem with this approach is that it violates a basic assumption. Testing must be done independently if it is to be fully effective in measuring software quality. Programmers are biased by the creative work they do. This blinds them to their own errors; it is human nature.

Approach 2. Testing is each unit's responsibility

Approach 2 fixes the obvious flaw encountered in approach 1 – that of programmers being inherently incapable of testing their own programs – by assigning product developers within the group the job of testing each other's code. Note from Figure 13.2 that each person is still wearing two hats: the person on the right is responsible for product development of their own modules, plus they must test their team-mate's modules.

The problem now is for these people to find the time to understand the job of the software testing professional as well as understand product development processes, standards, policies, tools, training, metrics, etc. For typical software industry projects, this is just asking too much of one person. It is like expecting your average construction site hire to be a master electrician and a master carpenter on the same project at the same time. It is not impossible, it can occur – it is just not likely and does not make much sense.

In reality, these people will pick one hat to wear – the hat for which they know they have the primary responsibility and for which they are evaluated by management – the hat of the product developer. They will take time to test other people's code as time and skills permit. They will usually not get very good at learning the special skills, tools, and methods of testing while they are simultaneously responsible for developing product. That's reality.



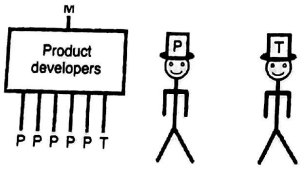
Advantages	Disadvantages
Solves the problem of programmers being incapable of testing their own code.	Programmer bandwidth (one hat too many).
	Methodologies, skills, training.

Figure 13.2 Approach 2: testing is each unit's responsibility.

Approach 3. Testing is performed by dedicated resource

Approach 3 solves the developer bandwidth issue by selecting a product developer and giving them a new job, that of test developer. Note from Figure 13.3 that each person in the group now only has to wear one professional hat. The tricky part here is for the group manager to pick the right person for testing.

In the evolutionary beginning, we noted that there was no test organization at all. One day the manager wakes up and declares, "All right, I see, I believe now. I know we need a dedicated person for testing. Hmmm, whom shall I tap to be our tester?" This really happens. Once while the author was presenting a course on testing, a development manager raised his hand and



Advantages	Disadvantages
Solves the developer bandwidth problem. (Only wearing one professional hat now).	Management bandwidth.
Single team.	Can management provide test process, standards, policies, tools, training, measures?

Figure 13.3 Approach 3: Testing performed by dedicated resource.

stated, "OK, OK, now I understand what testing is all about. I'm going to put our first full-time independent tester in place!" Great, thinks the course instructor, feeling very good about his teaching abilities.

"Yea", the manager continues, "I have this person that just doesn't fit on the development team. They've been a consistently poor performer; they're unhappy and they are just not contributing anything to reaching our project goals. I was going to fire them, but that is such an unpleasant task for me. I'll make them a tester! What have I got to lose?"

Of course we know he's got plenty to lose. The individual wearing too many hats now is the first line product development manager. This person, in addition to providing guidance to a product development team, must provide guidance on testing process, testing standards, testing policies, testing tools, testing training, testing measures, hiring professional testers, etc. It is just not going to happen.

It becomes clear that some sort of test group is necessary – a set of resources dedicated to performing testing activities and managed by a test manager. Without a formal organization, testing practices and tools must be set up for every project. With a separate test group, however, an organization remains in place to serve all projects on a continuing basis – to provide management with independent, unbiased, quality information.

Bill Hetzel, in the book *The Complete Guide to Software Testing* (1988), tells us:

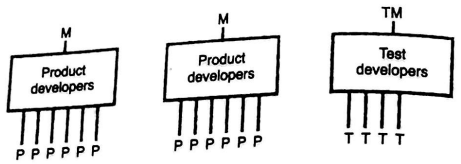
An independent test organization is important because

- building systems without one has not worked well,
- effective measurement is essential to product quality control,
- coordinating testing requires full-time, dedicated effort.

The creation of a formal test organization solves the problem of approach 3. Note in Figure 13.4 the importance of a test organization is realized, headed by a test manager. The question now becomes where to put the test group organizationally.

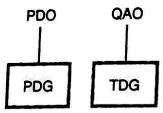
Approach 4. The test organization in QA

A common solution is to make the new test organization a component of quality assurance, where QA also audits the development process (see Figure 13.5). The manager of QA may not understand software testing. The capabilities of management of the testing group are critical, as is the manager of the testing manager. Since testing is performed in a separate organization from development, it will take extra effort to encourage and create a positive team environment. People also begin to ask, who owns quality? Development management complains that they no longer have the complete set of resources needed to produce a quality product.



Advantages	Disadvantages
Solves the management problems of approach 3.	Where to put it organizationally?

Figure 13.4 The test organization.



PDO = Product Development Organization
PDG = Product Development Group
QAO = Quality Assurance Organization
TDG = Test Development Group

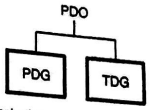
Advantages	Disadvantages
Solves the management problems of approach 3 by creating a formal test organization which is a part of QA.	Possible teamwork problems. TDG can get lost in QA group with other responsibilities. The product development group is not wholly accountable for producing quality products.

Figure 13.5 Approach 4: the test organization in QA.

Approach 5. The test organization in development

Approach 5 attempts to deal with the teamwork and quality ownership issue identified in approach 4 by creating a test organization that is part of the development organization (see Figure 13.6). This approach usually puts the test group under the second line product development manager.

Unfortunately, this is asking too much of most second line product development managers. It is much less an issue for higher level managers such as vice presidents that understand the need for a strong, independent



PDO = Product Development Organization
PDG = Product Development Group
TDG = Test Development Group

Advantages	Disadvantages
Solves the management problems of approach 3 by creating a formal test organization which is a part of development. Possibly solves teamwork issues of approach 4.	Does the senior development manager meet the test management requirements?

Figure 13.6 Approach 5: the test organization in development.

test function and who are willing to put a strong manager in place to manage the test group. In any event, all is riding on the senior manager who is now the one wearing two hats, that of managing the management responsible for product development and of managing the management responsible for software testing.

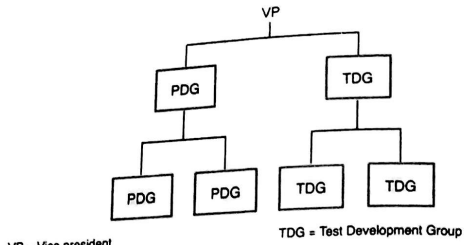
As organizations grow, more people are hired into the test organization, and multiple test groups are needed. A new issue arises. Should the multiple test groups be centrally or decentrally organized?

Approach 6. Centralized test organization

Approach 6 solves the senior management problem of approach 5 by creating a central test organization that lives within and serves a product development division (see Figure 13.7). Note how this creates a major opportunity for a senior test manager/director to significantly impact the organization. For example, the senior test manager can:

- manage the sharing of testing resources (both people and equipment) to smooth needs and better manage the risks of the company;
- coordinate consistent training for all testers across several test groups;
- promote consistent and high-quality testing tools for use by all testers;
- find and hire strong first line testing managers;
- provide real testing guidance to first line test managers.

In addition, first line testing managers see a potential career path in testing as they aspire to become a senior test manager.



Advantages	Disadvantages
Solves the senior management problems of approach 5 by centralizing the formal test organization which is a part of development.	Test organization lives or dies by VP.
Creates a career path for test managers.	Potential lack of teamwork at individual/project level.
	Possible lack of consistent test methodologies across VP organizations.

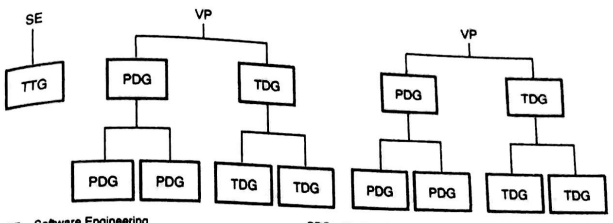
Figure 13.7 Approach 6: centralized test organization.

Now the risks are centralized, and assuming a good senior test manager is in place, the testing organization will live or die by the vice president who manages the product development and the test development organizations. When it's time to determine headcount, capital, and rewards for software testing, the vice president must provide the crucial support.

With the right vice president, this can and has often worked quite well in practice. A vice president that provides the proper resources, hires well, and delegates well to a highly competent senior test manager can make this approach work extremely well.

By creating project teams that are loaned resources from the independent test organization and that otherwise act, communicate, and live close together as a team alongside the product developers, the previously stated teamwork disadvantages are minimized. And when serious discussions regarding product quality are needed, a senior test manager is there to provide an unbiased voice to the vice president. This can be of great value to a vice president who cannot get the complete set of facts from product development management.

As companies continue to grow larger and contain multiple divisions, companies using this approach may begin to discover a lack of consistent approaches and best practices across different vice president organizations. This is solved by the next and final approach.



Advantages	Disadvantages
Solves the consistency problems of approach 6 by centralizing a test technology group which is a part of software engineering.	Test organizations live or die by VP.
	Potential lack of teamwork at individual/project level.

Figure 13.8 Approach 7: centralized and test technology.

Approach 7. Centralized test organization with a test technology center

Approach 7 solves the consistency problems of approach 6 by creating a test technology group, which in this case is part of a software engineering function (see Figure 13.8). This new test technology group is responsible for:

- leading and managing testing process and testing productivity improvement efforts;
- driving and coordinating testing training programs;
- coordinating the planning and implementation of testing tool programs;
- documenting test process, standards, policies, and guidelines as necessary;
- recognizing and leveraging best practices within the testing groups;
- recommending, obtaining consensus for, and implementing key testing measurements.

Selecting the right approach

How can we decide from among the many different approaches to using these structural design elements? To begin with, below is a set of sample selection criteria that can be used as a basis for evaluating different approaches. To what extent does the organizational structure:

- provide the ability for rapid decision making;
- enhance teamwork, especially between product development and testing development;
- provide for an independent, formal, unbiased, strong, properly staffed and rewarded, test organization;
- help to coordinate the balance of testing and quality responsibilities;
- assist with test management requirements as stated earlier in this chapter;
- provide ownership for test technology;
- leverage the capabilities of available resources, particularly people;
- positively impact morale and career path of employees (including managers)?

Providing for rapid decision making leads to improved responsiveness. It is possible to overcome an organizational structure that does not inherently provide for rapid decision making by creating special core teams and/or defining a special rapid escalation process, i.e., a daily high-level management short decision meeting used to escalate and deal with project issues.

One approach to doing reorganizations is to follow these steps:

(1) map the current organization;
(2) define and prioritize the key selection criteria (see list above);
(3) document new potential alternative approaches;
(4) evaluate potential approaches using a selection grid (see below);
(5) decide on a new organization;
(6) implement the new organization.

A decision matrix selection grid is a quality tool that can be used to evaluate the various approaches while using the selection criteria. Once the selection criteria are decided, they can be numbered or uniquely identified and placed across the top of the decision matrix.

The possible approaches are also numbered and placed vertically on the matrix as shown in Figure 13.9. Each approach is evaluated by considering each selection criteria and assigning a number to enter into the matrix. This number would come from a pre-defined range of values, e.g., from 1 to 5, where 5 indicates a high ranking, meaning the organization to a very great

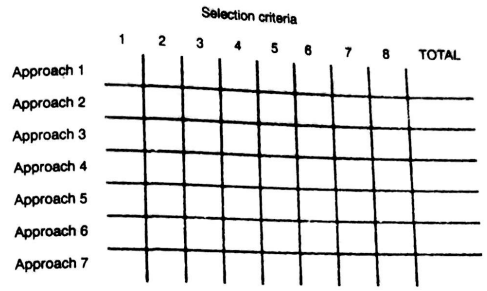


Figure 13.9 Decision matrix selection grid.

extent meets the given criteria for the approach under consideration. Likewise 3 might indicate a medium ranking and 1 a low ranking.

Add the scores horizontally to yield a total score for each approach. As a more advanced enhancement to this approach, the selection criteria can be assigned a multiplier weighting to be factored in.

Reorganizations should be carefully planned and implemented. One last piece of advice – remember to involve the participants in the above process!

References

Beizer, B. (1992). "Losing It, An Essay on World Class Competition," *Software Quality World*, 4(3).

Goodman, Paul, Sproull & Associates (1990). *Technology and Organizations*. San Francisco, CA: Jossey-Bass.

Gelperin, D. and Hetzel, W. (1989). *STEP: Introduction and Summary Guide*. Jacksonville, FL: Software Quality Engineering.

Hetzel, W. (1988). *The Complete Guide to Software Process*. Wellesley, MA: QED Information Sciences.

Humphrey, W.S. (1989). *Managing the Software Process*. Reading, MA: Addison-Wesley.

Kaner, C. (1988). *Testing Computer Software*. Blue Ridge Summit, PA: TAB Books.

Kit, E. (1992). "Approaches to Organizing the Independent Test Function," *Software Testing Analysis & Review (STAR) Conference Proceedings*.

Silverman, M. (1984). *The Technical Manager's Survival Book*. New York: McGraw-Hill.

Chapter 14
Current practices, trends, challenges

Advances in technology and development tools have enabled the creation of complex software using graphical user interfaces (GUIs) and client/server application architectures. Although these advances have improved development productivity, the resulting software is difficult to test, and the new demands on testing are eroding the development productivity gains achieved. Automation through commercial testing tools is stepping in to fill part of the gap.

During the short history of software engineering, there has been a dramatic shift in tester-to-developer ratios (in favor of the testers) which indicates that the importance of testing is now recognized, and organizations are now more willing to pay for proper testing. At the same time the number of testers continues to grow, and time spent on testing and its perceived effectiveness are increasing.

Meanwhile, studies of best practice in the industry indicate that the strictly non-technical basics – good management, communications, people, and controls – are more important to testing success than state-of-the-art methodologies, CASE tools, and other silver bullets.

GUIs: What's new here ?

There are two types of user interfaces:

(1) character-based user interfaces (traditional style);
(2) GUIs.

GUIs are characterized by the use of a different input device (mouse) and by high-resolution spatial and color images. GUI testing requires increased automation, but this is much more complicated in a GUI environment than a character-based environment. Fortunately, the new generation of capture/replay tools provides the answer (see Chapter 11).

Usage testing

Knowing how the product is really going to be used enables us to focus on certain types of critical areas when testing. When we prioritize validation tests, we are already, usually subconsciously, making judgments about potential usage. Usage testing is a more formal approach to this problem.

Usage testing is the process of:

- initial testing based on *estimated* usage patterns;
- measuring (collecting data on) the actual usage (after the product is complete and functional), and developing an *operational profile*;
- adjusting priorities, developing new tests, and retesting, based on the operational profile.

Usage testing reflects expected operational use in hardware and software configurations, in the frequency of operations tested, in the sequence of operations tested and in system load. Testing emphasis is on detecting errors that are the most likely to occur in operational use.

Usage testing is ideally performed after unit and integration testing, when substantial functionality is available, and after requirements validation – if it is performed separately. In practice it is usually performed after developing operational profiles based on real usage by real users, and this may mean that usage testing can apply only to releases after the first.

The advantages of usage testing are that it is customer oriented; test resources and schedules are optimized to maximize customer satisfaction. The hard part is cost-effectively obtaining operational profiles, which is why many companies today are not formally doing usage testing. However, most leading software companies do factor-in known or expected customer usage of the product when prioritizing tests worth developing.

Usage testing was pioneered by Frank Ackerman (1992, 1993), formerly of the Institute for Zero Defect Software.

Tester-to-developer ratios

Tester-to-developer staffing ratios are an interesting indication of the number of people performing testing activities compared to the number of people developing the software product. Ratios have improved dramatically in favor of more testers during the short and stormy life of the software engineering discipline.

Historically, for mainframes, the tester-to-developer ratio was 1:5–10, meaning one tester for every five to ten product developers. More recently published numbers include:

- Microsoft, 1992 2:3
- Lotus (for 1-2-3 Windows) 2:1
- Average of 4 major companies (1992) 1:2

(*Note*: The Microsoft and Lotus reference is Marvin (1993). The Microsoft ratio is for the Microsoft Worldwide Product Division. The four companies referenced are Microsoft, Borland, WordPerfect, and Novell for which the reference is Norman (1993).)

The above figures should be viewed as above average, that is, applying more testers than most companies do in practice. Informal surveys performed during testing courses taught by Software Development Technologies indicate that most companies do not enjoy ratios as good as one testing professional for every two product developers. More typical ratios are in the range from 1:3 to 1:4. Most testing professionals surveyed from companies operating at 1:5 or above (e.g., 1:7, 1:10, or even 1:20) generally felt that testing was under-resourced.

Software measures and practices benchmark study

This important study was a joint initiative by Xerox Corporation and Software Quality Engineering to improve understanding of software excellence. The goal of the study was to identify world-class projects in software technology and to characterize the engineering, management and measurement practices of these projects. A full research report is available from Software Quality Engineering (1991).

The design of the study included two phases. The purpose of phase 1 was to identify the "best" projects. Seventy-five software organizations were surveyed and responded to 52 questions (15 on the organization and 37 on its practices). The purpose of phase 2 was to characterize the practices of the "best" projects. Ten "best" projects (7 companies, 509 people) were surveyed, and responded to 104 questions (17 on attitude, 87 on practices and measures) and were subjected to an on-site practices assessment.

"Best" projects were selected on the following basis:

- perceived as producing high-quality results;
- perceived as using better practices;
- implemented recently or in final test with project team accessible for interview.

In other words, the best practices were reverse engineered from projects identified subjectively as having the highest quality – the projects the organization was the most proud of.

Companies were selected for phase 2 on the following basis:

- software a significant part of organizational mission;
- high scores on Phase 1 survey (above 75th percentile);
- reputation and public perception of success.

The actual phase 2 companies were:

- AT&T
- Dupont
- GTE
- IBM
- NCR
- Siemens
- Xerox.

The key findings of the study were:

(1) best projects emphasize strong up-front planning and close tracking and reporting of status on an ongoing basis;
(2) best projects rely on management fundamentals (teamwork, communication, and controls), not on technology and state-of-the-art methodologies;
(3) best projects emphasize reviews, inspections, and very strong and independent high-level testing;
(4) measurement used to track progress, quality problems, and issues.
(5) seven practices in common use related to planning and up-front requirements and design specifications;
(6) best projects utilized recommended practices – the investment required to achieve a more disciplined and structured process is evident in the results;
(7) best projects excel in different areas (survey categories) and emphasize different phases of the life cycle; no single project was superior in all areas.

The key measurement practices were discovered to be:

- schedule performance

- code defects
- test results
- test defects
- defects after release
- number of open problems
- time to correct problems
- issue tracking
- lines of code
- process compliance.

There are no silver bullets here. The keys to success are basic issues:

- good management
- good communications
- good people
- good controls
- ongoing measurements.

References

Ackerman, F. (1993). "Usage Testing," *Software Testing Analysis & Review (STAR) Conference Proceedings 1992.*

Ackerman, F. (1994). "Constructing and using Operational Profiles," *Software Testing Analysis & Review (STAR) Conference Proceedings 1993.*

Norman, S. (1993). "Testing GUIs is a sticky business," *Software Testing Analysis & Review (STAR) Conference Proceedings 1992.*

Software Quality Engineering (1991) Report 908.

Tener, M. (1993). "Testing in the GUI and Client/Server World," *IBM OS/2 Developer,* Winter.

Chapter 15
Getting sustainable gains in place

Only with a strong commitment to proven software engineering practices can companies successfully compete, survive, and thrive in today's marketplace. The necessary changes and investments to advance the state of the practice must be made; the alternative is to lose everything. Years of consulting experience, corroborated by the Xerox study (see Chapter 14), makes it clear that if we focus on planning, communication, tracking, and measurement, we'll substantially increase our odds of creating products with the right level of quality.

Getting gains to happen

Being an agent for change is hard. Whether you're striving for small or large changes, tactical or strategic improvements, those that require cultural evolution or revolution, take the time to assess the situation and plan for gains that are sustainable. Before proceeding, take a step back, look carefully around you. Are you facing small cracks in the pavement ahead or a deep crevasse? The strategy will be different for each. Taking the recurrent advice of making small steps in the name of continuous improvement when facing a large crevasse can be disastrous! Remember, as the Chinese proverb says:

"It is very dangerous to try and leap a chasm in two bounds."

Changing an organization is hard. Change must be positive, planned, and controlled, with a minimum of disruption, and a minimum of culture shock. Try to determine the readiness for change: How much pain is the organization feeling now? How much pain are our customers feeling now? What is the internal level of dissatisfaction with the status-quo versus the level of natural resistance to change (Migdoll, 1993)?

The benefits of change have to be defined, and the expected improvements justified. Change has to be sold to managers and practitioners throughout the organization. Measure the effectiveness of changes that have been